A Graph Coloring Approach to Dynamic Slicing of Object-Oriented Programs

Soubhagya Sankar Barpanda

(Roll No: 607CS005)

Department of Computer Science and Engineering

National Institute of Technology, Rourkela

Rourkela-769 008, Orissa, India

July, 2010.

A Graph Coloring Approach to Dynamic Slicing of Object-Oriented Programs

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Technology

(Research)

in

Computer Science and Engineering

by

Soubhagya Sankar Barpanda

(Roll No: 607CS005)

under the guidance of

Dr. Durga Prasad Mohapatra

Department of Computer Science and Engineering
National Institute of Technology, Rourkela
Rourkela-769 008, Orissa, India

July, 2010.

Dedicated to my parents

Department of Computer Science and Engineering
National Institute of Technology, Rourkela
Rourkela-769 008, Orissa, India.

Certificate

This is to certify that the work in the thesis entitled ***" A Graph Coloring Approach to Dynamic Slicing of Object-Oriented Programs"*** submitted by ***Soubhagya Sankar Barpanda*** is a record of an original research work carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Master of Technology (Research) in Computer Science and Engineering, National Institute of Technology, Rourkela. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Durga Prasad Mohapatra
Associate Professor
Department of CSE
National Institute of Technology
Rourkela-769008

Department of CSE
National Institute of Technology
Rourkela-769008

Place: NIT, Rourkela
Date: 26 - 07 - 2010

Acknowledgment

First of all, I would like to express my deep sense of respect and gratitude towards my advisor and guide Prof (Dr.) Durga Prasad Mohapatra, who has been the guiding force behind this work. I want to thank him for introducing me to the field of Program Slicing and giving me the opportunity to work under him. His undivided faith in this topic and ability to bring out the best of analytical and practical skills in people has been invaluable in tough periods. Without his invaluable advice and assistance it would not have been possible for me to complete this thesis. I am greatly indebted to him for his constant encouragement and invaluable advice in every aspect of my academic life. I consider it my good fortune to have got an opportunity to work with such a wonderful person.

Secondly, I would like to thank Prof. S. K. Jena, Prof. Banshidhar Majhi, Head CSE; Prof. S. K. Patra for serving on my Masters Scrutiny Committee.

I wish to thank all faculty members and secretarial staff of the CSE Department for their sympathetic cooperation.

During my studies at N.I.T. Rourkela, I made many friends. I would like to thank them all, for all the great moments I had with them, especially Mitrabinda, Narmada, Baikuntha and Vinay in the Software Engineering & Petrinet Simulation laboratory. I will be indebted to Baikuntha, who helped me a lot in tough periods.

When I look back at my accomplishments in life, I can see a clear trace of my family's concerns and devotion everywhere. My dearest mother, whom I owe everything I have achieved and whatever I have become; my beloved father, for always believing in me and inspiring me to dream big even at the toughest moments of my life; and my brother and sister; who was always my silent support during all the hardships of this endeavor and beyond.

Soubhagya Sankar Barpanda

Abstract

Program slicing is a decomposition technique, which produces a subprogram from the parent program relevant to a particular computation. Hence slicing is also regarded as a program transformation technique. A dynamic program slice is an executable part of a program whose behavior is identical, for the same program input, to that of the original program with respect to a variable of interest at some execution position. Dynamic slices are smaller than static slice, which can be used efficiently in different software engineering activities like program testing, debugging, software maintenance, program comprehension etc.

In this dissertation, we present our work concerned with the dynamic slicing of object-oriented programs. We have developed a novel algorithm, which incorporates graph coloring technique to compute dynamic slice of object-oriented programs. But in order to achieve the goal efficiently, we have contradicted the constraints of the traditional graph coloring theory. Moreover, the state restriction of the slicing criterion is taken into consideration, in addition to the dependence analysis. The advantage of our algorithm is that, it is more time efficient than the existing algorithms. We have named this algorithm, as *Contradictory Graph Coloring Algorithm* (CGCA).

Next, we extend our CGC Algorithm to compute the dynamic slice of concurrent and distributed object-oriented programs. We have taken the concurrent system dependence graph (CSDG) as the intermediate representation. The reason behind choosing the CSDG is that, it helps us represent correctly the concurrency aspects of object-oriented programs. While computing the dynamic slice, the control dependence, data dependence and synchronization dependence are taken into consideration including the state restriction of the slicing criterion. Similarly, we have used the distributed program dependence graph (DPDG) for computing the dynamic slice of object-oriented programs. The DPDG is capable of handling communication dependence that exists among different threads running on different machines. Hence, we have modified our CGC Algorithm to handle communication dependence among different threads.

The advantage of our algorithm is that, the state restriction of the slicing criterion is considered along with dependence analysis. Moreover, the proposed algorithm can be made faster by applying any off-the-shelf optimization technique like heuristic method, probabilistic method etc.

Dissemination of Work

Published

1. S. S. Barpanda, B. N. Biswal and D. P. Mohapatra, **Contradictory graph coloring algorithm to compute slice of concurrent object-oriented programs**, *International Journal of Computational Vision and Robotics*, Inderscience Publications, England. (In Press)

2. S. S. Barpanda, B. N. Biswal and D. P. Mohapatra, **A Graph Coloring Technique to Slicing of Object-Oriented Programs**, *1st International Conference and Workshop on Engineering and Technology (ICWET-2010)*, ACM, February 2010, Mumbai, India.

Communicated

1. S. S. Barpanda, B. N. Biswal, D. P. Mohapatra, **Dynamic Slicing of Distributed Object-Oriented Programs**, *Communicated to IET Software*.

Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Description
PDG	Program Dependence Graph
SDG	System Dependence Graph
CSDG	Concurrent System Dependence Graph
DPDG	Distributed System Dependence Graph
PDN	Program Dependence Net
ClDG	Class Dependence Graph
DODG	Dynamic Object-Oriented Dependence Graph
CGCA	Contradictory Graph Coloring Algorithm

Chapter 1

Introduction

Program slicing [113] is regarded as a program analysis technique. It is a decomposition technique that extracts program statements from the parent program relevant to a particular computation. Program slices are computed with respect to a slicing criterion. A *slicing criterion* is typically a pair, that consists of statement number s and a set of variables V represented as $< s, V >$. Program slicing can be used effectively in various software engineering activities [27, 95, 114, 116] like testing, debugging, software maintenance, program comprehension etc.

Mark Weiser [110] was the first to innovate the concept of program slicing as he observed that programmers have some abstractions about the program in mind while debugging. Since the innovation of program slicing, it has undergone many refinements. Weiser defined technique deals with producing program slice S by simply deleting those statements from the parent program P which don't influence other parts of the program P such that S replicates the behavior of P. This slicing technique originally introduced by Weiser [110, 111] is also called static backward slicing. It is static because the slice is computed irrespective of any input values to the program and backward because the control flow of the program is considered in reverse while computing the slice.

Now-a-days, object-oriented programming languages have dominated the software market. The advantages of object-oriented programs over structured programs include data security, data abstraction, re-usability etc. Moreover, many of these real world object-oriented programs are very large in size and concurrent in nature. In addition, object-oriented programs like Java are usually preferred for developing enterprise

level solutions. These applications run in a distributed manner on several nodes connected through a network. The size of these programs often exceeds million of lines. Developing such large programs which should be reliable, is a challenging job. At this point program slicing can be used to ease the job. Program slicing is used as a technique to analyze a program. The results of the analysis can be used to help in debugging, test case design, test coverage analysis etc.

Slicing of object-oriented programs poses new challenges which are not encountered in slicing of structured programs. The reason behind it is that, object-oriented programs have some special features which are not available in the traditional structured programs. Those features include class and object, dynamic binding, encapsulation, inheritance and message passing etc. Due to the above said features available in object-oriented programs, the process of tracing dependencies becomes more complex than that in a procedural program. Larson and Harrold [74] were the first to consider the slicing of object-oriented programs. To address these object-oriented features, they had extended scope of the system dependence graphs (SDG) [58] to represent object-oriented software. After the SDG is constructed, the two phase algorithm of Horwitz et al. [58] is used with minor modifications for computing static slices. But their approach was to compute the static slice of an object-oriented programs, and did not address dynamic slicing aspects. The dynamic slicing aspects have been reported by Zhao [121], Song et al. [101], Xu et. al. [114] and Wang et al. [107].

It is known that after the evolution in enterprise level software, the software development process is mainly focused on developing concurrent and distributed software. But testing and maintenance of these softwares is a cumbersome job as well as expensive. At this point slicing techniques can be used to ease the different software development activities. An extensive literature survey reveals that, most of the research work in program slicing are dedicated towards sequential programs. There are very few research reports [25, 66, 67, 123] available, which deal with the slicing of concurrent and distributed programs.

A major goal of any slicing technique is efficiency. Dynamic slicing is more preferred to static slicing as the results obtained from the analysis is found to be very useful in

the testing phase of software development. Efficiency is a matter of concern in the case of slicing of object-oriented programs, since they are of large sizes. If the slicer is very slow in producing the slice, then it will be useless. Hence the program is analyzed first using a proper intermediate representation, from which the slice is computed using an efficient algorithm.

1.1 Categories of Program Slicing

Program slicing [54] has been categorized by taking into consideration the run-time environment, graph traversal and the program being from which slices are computed. Below, we have given a brief description about the different types of program slice, which have been found in literature survey.

Static Slicing and Dynamic Slicing:

In static slicing [58, 34, 46], slices from the parent program are computed irrespective of any input. So, the slice are computed for all possible values of the input variables. Static slices are less useful as they contain most of the statements of the parent program.

Korel and Laski [2, 36, 64, 118] were the first to give the notion of dynamic slicing. Dynamic slices [65, 89] are computed from the parent program for a particular value of the input variable during execution. Therefore, dynamic slices are smaller than static slice. Dynamic slices are very helpful in testing and maintenance phase of software development process.

Backward Slicing and Forward Slicing:

A backward slice [111, 58] contains all parts of the program that might directly or indirectly affect the slicing criterion. Thus a static backward slice provides the answer to the question: "*which statements affect the slicing criterion?*".

A forward slice [101, 118] with respect to a slicing criterion contains all parts of the program that might be affected by the variables used or defined at the program point of consideration. A forward slice provides the answer to the question: "*which state-*

ments will be affected by the slicing criterion?". Unless otherwise specified, we consider backward slices throughout this thesis.

Intra-procedural Slicing and Inter-procedural Slicing:

Intra-procedural slicing [58, 39] computes slices within a single procedure. Calls to other procedures are either not handled at all or handled conservatively. If the program consists of more than one procedure, inter-procedural slicing can be used to derive slices that span multiple procedures.

For object-oriented programs, intra-procedural slicing is meaning less as practical object-oriented programs contain more than one method. So, for object-oriented programs, inter-procedural slicing is more useful.

1.2 Motivation of Our Work

A major goal of any dynamic slicing technique is to produce as small a slice with respect to a slicing criterion as possible since smaller slices are found to be more useful for different applications. Much of the literature on program slicing is concerned with improving the algorithms for slicing in terms of reducing the size of the slice and improving the efficiency of the slicing algorithm.

As discussed earlier, now-a-days most of the programs are object-oriented in nature which are quite large and complex. Hence, it is difficult to test and maintain these products. Program slicing techniques have been found to be useful in applications such as program understanding, debugging, testing, software maintenance and reverse engineering. Particularly dynamic program slicing is used in interactive applications such as debugging and testing of programs. Therefore the dynamic slicing techniques need to be efficient. There are few research reports [24, 26, 77] available for slicing of object-oriented programs which are less efficient. Therefore, there is a dire necessity to develop efficient algorithms for dynamic slicing of object-oriented programs. Moreover, most of the object-oriented programs are concurrent and distributed in nature. This poses a challenge to the present software engineers to find out suitable techniques

for developing error free software for concurrent and distributed environment. Dynamic slicing techniques guarantee efficiency at this end. However, research attempts in the program slicing area have focused attention largely on sequential programs. But, research reports dealing with slicing of concurrent and distributed programs are scarce [30, 33, 34, 37, 62, 63, 76] in the literature.

With this motivation for developing techniques for dynamic slicing of object-oriented programs, concurrent and distributed object-oriented programs, we identify the major goals of this thesis.

1.3 Objective of Our Work

Based on the motivation outlined in the previous section, the main objective of our research work is to develop an efficient algorithm to compute the dynamic slice of object-oriented programs. More specifically, the main objectives of the study is as follows:

- To develop an efficient dynamic slicing algorithm for object-oriented programs, using a proper intermediate representation.
- Next, we wish to extend this approach to compute dynamic slice of concurrent object-oriented programs.
- Slicing of distributed object-oriented programs is very cumbersome because it is usually large in size and run on different machines connected through a network. Thus, we aim to extend our proposed algorithm to compute slice of distributed object-oriented programs.

1.4 Organization of The Thesis

The rest of the thesis is organized as follows:

Chapter 2 provides the background concepts used in the rest of the thesis. We also discuss some graph-theoretic concepts and definitions. We describe some intermediate program representation concepts, which are used in slicing techniques and used later

in our algorithms. Then, we briefly present some basic applications of program slicing, which are found helpful in software development process.

Chapter 3 provides a brief literature review. First, we provide a brief discussion about dynamic slicing of object-oriented programs. Then, we discuss the reported work on slicing of concurrent object-oriented programs. Finally, we discuss the reported work on dynamic slicing of distributed programs.

Chapter 4 first provides a brief literature survey of slicing of object-oriented programs followed by our proposed approach to compute dynamic slice using graph coloring techniques.

Chapter 5 presents basic concepts of concurrency models in Java followed by our proposed algorithm which is extended to compute dynamic slice of concurrent object-oriented programs.

Chapter 6 discusses about the dynamic slicing of distributed object-oriented programs. First, the distributed nature of Java is presented along with a brief literature survey. Then, an extended version of the proposed algorithm is used to compute dynamic slice of distributed object-oriented programs.

Chapter 7 presents the concluding remarks, with scope for further research work.

Chapter 2

Basic Concepts and Terminologies

The area of program slicing has got maturity over the last two decades by contributions from several researchers. Since its innovation many new concepts have been introduced to extend the horizon of program slicing. Many faster algorithms have been proposed so as to speed up the process of slicing and to make program slicing usable in different applications. Today the slicing technique is being applied to object-oriented programs, because object-oriented programs are now-a-days more popular in software market. Also, these slicing techniques have been applied to diverse problem areas. Some of the important applications of slicing have been discussed in Section 2.4 of this chapter.

This chapter provides a gist of the background used in the rest of this thesis. For the sake of conciseness, we do not aim to present a detailed description of the background theory. Instead, we provide a brief introduction aimed at highlighting the basic concepts and definitions. The basic concepts and definitions are used in subsequent chapters of this thesis. Section 2.1 describes some intermediate program representation concepts which are commonly used in slicing techniques. Section 2.2 briefly discusses about program projection theory. Section 2.3 discusses the basic concepts of graphs coloring. In Section 2.4, some important applications of program slicing are given, which are found useful in different day-to-day software development activities. Section 2.5 summarizes this chapter.

2.1 Program Representation

Taking into consideration the exact context of use, different program representation schemes have been proposed. These include pseudo-code, machine level instruction, flow charts etc. A proper program representation [68] helps in computing slice efficiently. In the following, a few basic concepts associated with intermediate program representations is presented that are used later in this thesis.

2.1.1 Control Dependence Graph

Control dependence graph was first introduced by Ferrante et al. [38] to represent the relations between program entities that arise due to control flow.

Control Dependence: If G be the Control flow graph of a program P and x and y be two nodes in G, then node y is control dependent on a node x if and only if there exists a directed path D from x to y, y post-dominates every z in D (excluding x and y) and y does not post-dominate x.

Control Dependence Graph: The control dependence graph of a program P is the graph $G = (N, E)$, where each node $n \in N$ represents a statement of the program P and $(x, y) \in E$ if x is control dependent on y.

2.1.2 Program Dependence Graph

Ferrante et al. [58, 38] presented a new mechanism of program representation called Program Dependence Graph (PDG). An important feature of PDG is that it explicitly represents both control and data dependencies in a single program representation. A PDG models a program as a graph in which the nodes represent the statements, and the edges represent inter-statement data or control dependencies.

Program Dependence Graph (PDG): The program dependence graph G of a program P is the graph $G = (N, E)$, where each node $n \in N$ represents a statement of the

program P. The graph contains two kinds of directed edges: control dependence edges and data dependence edges. A control (or data) dependence edge (m,n) indicates that n is control (or data) dependent on m.

2.1.3 System Dependence Graph

The PDG of a program represents the control dependencies and the data dependencies in a single framework. The PDG is actually a suitable intermediate representation for intraprocedural slicing. But, it cannot handle procedure calls. Horwitz et al. [58] added new features to the PDG representation to facilitate inter-procedural slicing. They introduced the System Dependence Graph (SDG) representation which models the main program together with all associated procedures.

The SDG represents a program in a language with the following properties:

- A complete program consists of a main program and a collection of auxiliary procedures.
- Procedures end with return statements. A return statement does not include a list of variables.
- Parameters are passed by value-results.

The SDG is equivalent to the PDG in terms of program representation. Indeed, a PDG is a subgraph of the SDG. In other words, for a program without procedure calls, the PDG and SDG are identical. The technique for constructing an SDG consists of first constructing a PDG for every procedure, including the main procedure, and then adding auxiliary dependence edges which link the various subgraphs together. This results in a program representation which includes the information necessary for slicing across procedure boundaries.

A system dependence graph also abbreviated often as SDG is nothing but a collection of procedure dependence graphs, each representing one procedure as a graph. Larsen and Harrold [74] have developed their own SDG so as to accommodate the need of object-oriented programs. Their proposed representation helps model both

complete as well as incomplete systems. Below, we give a brief discussion about how to use SDG to represent the above two systems.

Incomplete systems:- In order to facilitate analysis, it is needed to represent individual class of an object-oriented software. It is done with the help of class dependence graph (ClDG) [74]. A ClDG captures the control and data dependence relationships that can be determined about a class without knowledge of the calling environment. The methods in a ClDG are represented using the same procedure dependence graph. Each method has a method entry vertex that represents the entry into the method. A ClDG also contains a class entry vertex that is connected to the method entry vertex for each method in the class by a class member edge. Class entry vertices and class member edges let us quickly access method information when a class is combined with another class or system. Our ClDG construction expands each method entry by adding formal-in and formal-out vertices. We add formal-in vertices for each formal parameter in the method, and formal-out vertices for each formal reference parameter that is modified by the method. Additionally, we add formal-in and formal-out parameters for global variables that are referenced in a method. Finally, since a class's instance variables are accessible to all methods in the class, we treat them as globals to methods in the class, and we add formal-in and formal-out vertices for all instance variables that are referenced in the method. The exception to this representation for instance variables is that our construction omits formal-in vertices for instance variables in the class constructor and formal-out vertices for instance variables in the class destructor.

Complete systems:- We construct the SDG for a complete program by connecting calls in the partial system dependence graph to methods in the ClDG for each class. It involves connecting call vertices to method entry vertices, actual-in vertices to formal-in vertices, and formal-out vertices to actual-out vertices. The summary edges for methods in a previously analyzed class are added between the actual-in and actual-out vertices at call sites. This construction of the SDG for an object-oriented system maximizes reuse of previously constructed portions of the representation. The intro-

duction of variables in the scope of the application program, such as a global variable, does not affect the representation in any of the ClDG's. Any global variables referenced or modified by a class must be declared extern in the class, so this information would have been included while building the class's ClDG.

2.1.4 Concurrent System Dependence Graph

The concurrent system dependence graph proposed by Mohapatra et al. [87] is an ideal intermediate representation for concurrent object-oriented programs. When inter-thread synchronization and communication are present, some control and data flows in the threads of a concurrent Java program become interdependent. This aspect of concurrency is captured successfully by the CSDG. We will use the CSDG to compute dynamic slice of concurrent object-oriented programs. The CSDG can capture the program dependences that can be determined statically as well as at run-time.

Concurrent Control Flow Graph (CCFG): A *concurrent control flow graph* (CCFG) G of a program P is a directed graph (N, E, Start, Stop), where each node $n \in N$ represents a statement of the program P, while each edge $e \in E$ represents potential control transfer among the nodes. Nodes *Start* and *Stop* are two unique nodes representing entry and exit of the program P, respectively. There is a directed edge from node a to node b if control may flow from node a to node b.

Synchronization Dependence: A statement y in one thread is synchronization dependent on a statement x in another thread if execution of y is dependent on execution of x due to a synchronization operation.

Communication Dependence: There are two types of communication dependencies in Java. In the first one, communication among different threads may be established through sockets and using constructs like *getOutputStream()* and *getInputStream()*. We have named this type of communication dependence M-Communication dependence. In the second one, Java uses shared memory to support communication among threads.

In this type of communications, two threads executing in a parallel manner may exchange their data via shared objects. This type of communication dependence is named as S-Communication dependence.

2.1.5 Distributed Program Dependence Graph

Distributed Program Dependence Graph is an intermediate representation to represent distributed object-oriented programs. The notion of DPDG was first given by Mohapatra et al. [88]. The beauty of the graph is that, it helps in capturing concurrency that exists among different threads running on different machines. A logical node is used in DPDG, to represent communication dependency among threads running on different machines.

Distributed Control Flow Graph (DCFG): A distributed control flow graph (DCFG) G of a component program p_i of a distributed program $P(p_1,\ldots,p_n)$ is a flow graph, where each node represents a statement of p_i, and each edge $e \in E$ represents potential control transfer among the nodes. Nodes *Start* and *Stop* are two unique nodes which represent the entry and exit nodes of the component program P_i respectively. There is a directed edge representing a control flow from node a to node b iff there is a control flow from node a to node b.

Thread Dependence: For a DCFG G, if x be the node representing the *run*() statement of thread p_i and node y is said to be thread dependent on x, iff there exists a directed path from x to y such that none of the nodes in that path is a *run* node.

Communication Dependence: In a Java program two types of communication dependencies may exist. We restrict communication dependency among threads belonging to the same component program to be only S-Communication dependence type. Whereas communication dependency among threads belonging to different component programs is termed as M-Communication dependence type. In S-Communication dependence, shared memory may be used to support communication among threads.

In this type, two threads exchange data via shared objects. In M-Communication dependence, communication among threads occurs through sockets.

2.2 Formalization of Program Slicing

Binkley et al. [20] have proposed a system called *Program Projection Theory* so as to formalize the previously developed frameworks for program slicing. The program projection theory provides a general framework to formulate different slicing approaches. One of the beauties of the proposed theory is to capture and define the semantics preserved by slicing algorithms. This helps in proving the correctness of an algorithm. Below we are giving few important notations of program projection theory.

[$(\lesssim, \approx)$ - projection]: If $\lesssim$ be a pre-order over programs and $\approx$ be an equivalence relation over programs, then program q is a $(\lesssim, \approx)$ - projection of p iff $q \lesssim p$ and $q \approx p$.

[syntactic ordering: $\sqsubseteq$]: If F be a function that takes a program P as input and returns a partial function from line number l to statement c, then program p contains the statement c at line number l . Hence, we write

$$p \sqsubseteq q \Leftrightarrow F(p) \subseteq F(q)$$

[state trajectory]: It is a finite sequence of pairs consisting of line number and state and represented as $(n_1, \sigma_1)(n_2, \sigma_2)\ldots(n_k, \sigma_k)$, where n_i represents statement executed at that line number and σ_i represents the current state of the slicing criterion after the execution of the statement.

[State restriction, ($\restriction$)]: If σ be the state and V be the set of variables,then $\sigma \restriction V$ restricts σ so that it is defined only for variables of V

$$(\sigma \restriction V) = \begin{cases} \sigma x & \text{if } x \in V \\ \perp & \text{otherwise} \end{cases} \tag{2.1}$$

Danicic et al. [99] have introduced a new non-strict semantics for a simple while

language. Their new semantics allows to give a denotational definition of variable dependence and *neededness*, which is consistent with program slicing. Finally, our semantics is proved to be preserved by slicing algorithms, which makes it very useful to prove correctness of slicing algorithms. Furthermore, they have shown that their new semantics is substitutive. This property is very useful in proving correctness of program transformations.

Barraclough et al. [11] introduced a new, substitutive, intuitive, finite trajectory-based semantics of programs. They have proved that its induced equivalence is a natural extension, to non-terminating programs, of the equivalence introduced by Weiser that only considers terminating programs. They have also proved that, slicing algorithms based on traditional data and control dependence preserve, thereby showing that the new semantics captures the behavior of program slicing algorithms for both terminating and nonterminating programs. They have considered concrete programs, but it is well known that program schemas equivalence classes of programs contain all the necessary information to formally investigate dependence-based slicing algorithms.

2.3 Coloring of Graphs

In this Section, a brief discussion about graph coloring technique is given. Also, we have tried to explore the application of graph coloring. Coloring of graphs is one of the classical problem in graph theory. Since, a graph has two basic components i.e. *node* and *edge*; so coloring problem can be classified as either *node coloring* or *edge coloring*. Below, we have given a gist about node coloring, which we will use to compute the dynamic slice of object-oriented programs, discussed later in the thesis.

In graph theory [35], node coloring is known to be a very robust technique, which is a special case of graph labeling. It deals with assigning *colors* to the nodes subject to certain constraints. The coloring process is carried out by ensuring that "no two nodes sharing the same edge have the same colors". Coloring of graph using at most *k-colors* is called k-coloring. The smallest number of colors required to color a graph *G* is called

its chromatic number denoted by $\chi(G)$. A graph is k-chromatic if its chromatic number is exactly k. Coloring technique can be broadly divided into two categories defined below:

- **Strong coloring:-** Coloring all the nodes of a graph with colors such that no two adjacent nodes have the same color is called the strong coloring of a graph, also known as proper coloring of graph. A graph in which every node has been assigned a color according to proper graph coloring is called properly colored graph. Actually, a given graph can be colored in many different ways.
- **Weak coloring:-** In weak coloring of a graph [69, 6, 91], a color is assigned to each node, such that each non-isolated node is adjacent to at least one node with different colors.

2.4 Applications of Program Slicing

This Section describes the use of program slicing techniques in various applications. The program slicing technique was originally developed to realize automated static code decomposition tools. The primary objective of those tools was to aid program debugging. From this modest beginning, the use of program slicing techniques has now ramified into a powerful set of tools for use in different software development processes.

2.4.1 Differencing

Novice programmers usually find it difficult to differentiate two programs. Program slicing can be used effectively to differentiate between two programs [18]. There are two related differencing problems:

1. To find all the components of two programs having different behavior.
2. To produce a program that captures the semantic differences between two programs.

For programs *old* and *new*, a simple solution to problem 1 is obtained by comparing the backward slices of the vertices in *old* and *new's* dependence graphs G_{old} and G_{new}. Here, the backward slice is computed with respect to a given slicing criterion. Components whose vertices in G_{new} and G_{old} have isomorphic slices have the same behavior in *old* and *new*; thus the set of vertices from G_{new} for which there is no vertex in G_{old} with an isomorphic slice approximates the set of components *new* with changed behavior.

A solution to the second differencing problem is obtained by taking the backward slice w.r.t. the set of affected points (i.e., the vertices in G_{new} with different behavior than in Gold). For programs with method calls, two modifications are desired: First, inter-procedural slicing techniques are required to be used to ensure that the resulting program slice is executable. Second, this solution is overly pessimistic: consider a component c in method P that is invoked from two call-sites $c1$ and $c2$. If c is identified as an affected point by a forward slice that enters P through $c1$ then we will include $c1$ but not $c2$ in the program that captures the differences. However, the backward slice with respect to c would include both $c1$ and $c2$.

2.4.2 Debugging

The problem of finding bugs in a program is always a tedious job. The process of finding a bug involves in running the program several times, learning more and narrowing down the search each time, till the bug in the code is finally located. In distributed systems, the problem is more difficult because of control and data dependencies and also communication dependencies that might lead to additional, often non-repeatable bugs. Program slicing was originally proposed by observing the process of debugging carried out by programmers [111, 33, 80, 109]. Programmers mentally compute slice [2] while debugging codes. Even after several advancements to the basic slicing techniques, program debugging remains a key application area of slicing techniques.

2.4.3 Software Maintenance

Software maintenance is a costly process because each modification to a program must take into account many complex dependence relationships in the existing software.

The challenges in software maintenance, are to understand various dependencies in an existing software and to make changes to the existing software without introducing new errors. One of the problems in software maintenance is that of the ripple effect, i.e., whether a code change in a program will affect the behavior of other codes of the program. To avoid this problem, it is important to know which variables will be affected by a modified variable. This problem can be greatly reduced by slicing the software being maintained [41, 15, 16].

2.4.4 Testing

Software maintainers often carry out regression testing. Regression testing deals with re-testing of software after modification [27, 116, 16, 40, 47, 53, 81, 51]. Even after the smallest change to a piece of code, extensive tests may be necessary which might involve running a large number of test cases to ensure that no unwanted behavior arises due to the change. This requires to generate new test cases along with the existing test cases. Slicing can be used to reduce the number of these test cases. While decomposition slicing eliminates the need for regression testing on the complement, there may still be a substantial number of tests to be run on the dependent, independent and changed parts. Slicing can be used to reduce the number of these tests.

2.4.5 Program Integration

Programmers many times face the problem of integrating several variants of a base program. The first step is to look for textual differences. More sophisticated techniques for this purpose have now become available. Semantics-based program integration is a technique that can create an integrated program that incorporates the changed computations of the variants as well as the computations of the base program that are preserved in all variants [19, 57]. Horwitz et al. [47] presented an algorithm for semantic-based program integration that creates the integrated program by merging certain program slices of the variants. Their integration algorithm takes as input three programs, Base, A and B, where A and B are variants of Base. The integrated program is produced by (1) building graphs that represent Base, A and B, (2) combining program

slices of the program dependence graphs of Base, *A* and *B* to form a merged graph, (3) testing the merged graph for certain interference criteria, and (4) reconstructing a program from the merged graph. Yang extends the algorithm of Horwitz et al. for detecting program components with equivalent behaviors and it can accommodate semantics preserving transformations.

2.4.6 Functional Cohesion

Cohesion measures the relatedness of the code of some component [29]. A highly cohesive software component is one that has only one function which can't be further divided into sub-modules. Bieman and Ott [14] define data slices to consist of data tokens (instead of statements). Data tokens may be variables of constant definitions and references. Data slices are computed for each output of a procedure (e.g., output to a file, output parameter, assignment to a global variable). The tokens that are common to more than one data slice are the connections between the slices. They are called glue. The glue binds the slices together. The tokens that are in every data slice of a function are called super-glue. Strong functional cohesion can be expressed as the ratio of super-glue tokens to the total number of tokens in the slice, whereas weak functional cohesion may be seen as the ratio of glue tokens to the total number of tokens. The adhesiveness of a token is another measure expressing how many slices are glued together by that token. Dallal [5] has conducted an experimental comparison study using five software applications to show the effectiveness of the Computing-All-Slices algorithm in computing the functional cohesion of program functions.

2.4.7 Other Applications of Program Slicing

Program slicing techniques have been used in diverse fields [95, 41, 29] which include compiler optimizations, detecting dead code, software portability analysis, program understanding, program verification [60], measuring class cohesion [14], reverse engineering [12] etc.

2.5 Summary

In this chapter, we have discussed some definitions and concepts that will be used later in our algorithms. We have also discussed various intermediate program representations. We discussed some basic concepts of static slicing which are required to understand our dynamic slicing techniques. Finally, we provided an overview of some important applications of program slicing.

Chapter 3

Literature Review

In this chapter, we present a survey on slicing of object-oriented programs. A strong review of literature reveals that, there is a scarcity [24] of research papers describing the slicing of object-oriented programs. Moreover, research papers dedicated towards dynamic slicing of object-oriented programs are very rarely found. We have also provided a thorough survey of slicing of concurrent object-oriented programs. Then, we have discussed the existing work on slicing of distributed object-oriented programs.

3.1 Dynamic Slicing of Object-Oriented Programs

Ohata et al. [93] have proposed a unique program slicing method for object-oriented programs to evaluate its effectiveness with Java programs. They [93] have adopted an intermediate slicing method, that lies between static and dynamic slicing. They have named the new technique as *Dependence-Cache* (DC) slicing to object-oriented programs. DC slicing method uses dynamic data dependence analysis and static control dependence analysis. The DC slice is computed in four phases. In phase 1, the defined variables and referred variables for each statement are identified. In phase 2, the data dependence and control dependence relations are extracted between program statements. In phase 3, the program dependence graph (PDG) is constructed using the dependence analysis done in phase 2. In phase 4, the PDG is traversed backward to compute slice with respect to the slicing criterion specified by the user. They have proposed an algorithm, named as *Object-Oriented Dynamic Cache algorithm*. The size of OODC slice is claimed to be 20% - 70% as large as that of static slice, hence more precise than static slice. But the DC slices are found to be larger than dynamic slice.

The DC slice has two advantages. It helps us to solve *array indices problem* and *pointer alias problem*.

Xu and Chen [26] have proposed a dynamic slicing method for object-oriented program based on dependence analysis. They [26] have proposed an algorithm to compute dynamic slice by combining static dependence information and dynamic execution of the program. To represent the different features of object-oriented programs like class, object, dynamic binding, inheritance etc., they have extended the notion of system dependence graph (SDG) given by Horwitz et al. [58]. They have named the new graph as object *program dependence graph* (OPDG). While computing dynamic slice from *OPDG*, they have considered three basic structures: *sequence*, *branch* and *loop*. By analyzing the control flow graph, fewer breakpoints are inserted to trace the execution of the program. It is an approach combining forward analysis with backward one. In the forward process, it marks information on the object program dependence graph (OPDG) and computes dynamic slices (they are used to record dynamic execution information) at the necessary points during the program execution. In the backward process, it traverses the OPDG marked to obtain the final dynamic slices. Based on this model they have proposed methods to dynamically slice methods, objects and classes. Moreover, their proposed algorithms focus on method slicing, object slicing and class slicing. Their [26] algorithms are found to be useful in localizing errors when debugging. The disadvantage of their method is that, when the source program is sliced, it is too much slow, because all the methods are analyzed first before slicing and results are stored in libraries on disk.

Song and Huynh [101] have proposed a slicing technique to compute forward dynamic slice of object-oriented programs. They have proposed an algorithm to slice object-oriented programs with respect to a slicing criterion that recursively decomposes constructors and member functions. According to their approach, they decompose the instance variables of each object beginning by slicing its constructors. They decompose the inheritance hierarchy of object-oriented programs with respect to instance variables. The resulting slices are reused while the instance variables remain untouched. The behavior of each variable is tracked in each constructor such that,

when any two decomposed portions are merged, the positions of the parameters are identified. They have also proposed a method to compute slice from functions, in the presence of recursive function calls. They have also developed a dynamic object relationship diagram (DORD) that shows the interaction of objects with respect to specified variables.

Wang et al. [107] presented a new dynamic slicing algorithm for Java programs which operates on compact byte code traces. According to their algorithm, the byte code stream corresponding to an execution of a Java program is compactly represented. Then, they perform a backward traversal of the compressed program trace to compute data / control dependencies on-the-fly. The slice is updated as these dependencies are encountered during the traversal. Wang and Roychoudhury [108] have developed a dynamic slicing method for Java programs. Their technique proceeds by backwards traversal of the bytecode trace produced by an input I in a given program P. They have used results from data compression to compactly represent bytecode traces, beacuse such traces can be huge. The major space savings in their method came from the optimized representation of data addresses used as operands by memory reference bytecodes, and instruction addresses used as operands by control transfer bytecodes. They have also extended their dynamic slicing algorithm to perform *relevant slicing*.

Zhao [121] has modified the dynamic dependence graph (DDG) proposed by Agrawal and Horgan [3]. The modified graph is known as dynamic object-oriented dependence graph (DODG) to represent various dynamic dependencies between statement instances for a particular execution of an object-oriented program. The DODG is an arc-classified diagraph (V,A), where V is the multi-set of flowgraph vertices, and A is the set of arcs representing dynamic control dependencies and data dependencies between vertices. The construction of the DODG is based on dynamic analysis of control flow and data flow of the program, and it is similar to those for constructing dynamic dependence graphs for procedural programs. They have constructed the DODG by creating a new node for each occurrence of a statement in the execution history, and creating all the dependence edges associated with the occurrence at run-time. He has also considered the specific features of object-oriented programs such as method calls,

inheritance and polymorphism. Based on the DODG, Zhao has used a two-phase algorithm to compute dynamic slices of object-oriented programs. Computation of dynamic slices using the DODG is carried out as a graph-reachability problem. The disadvantage of this approach is that the number of nodes in a DODG is equal to the number of executed statements, which may be unbounded for programs having many loops.

Mohapatra et al. [86] have proposed two novel dynamic slicing algorithms for object-oriented programs. One is known as edge marking dynamic slicing (EMDS) algorithm and the other one is known as node marking dynamic slicing (NMDS) algorithm. They have proposed a new dependence graph called *extended system dependence graph* (ESDG) as the intermediate representation. The EMDS algorithm is based on marking and unmarking the edges of the ESDG as when the dependencies arise and cease during run-time. The NMDS algorithm is based on marking and unmarking the executed nodes of the ESDG appropriately during run-time. The advantage of both the algorithms compared to the related ones is that they do not not require any new nodes to be created and added to ESDG at run-time nor do they require to maintain any execution trace in trace files. The limitation of their approach is that, it takes considerable time in marking and unmarking of edges or nodes during the entire process.

3.2 Slicing of Concurrent Object-Oriented Programs

Several research reports are found during review of literature, which are dedicated towards static slicing of concurrent object-oriented programs [95, 25, 66, 123, 26, 44, 31, 122, 125]. There are very few research reports, which deal with dynamic slicing of concurrent object-oriented programs [87, 32].

Zhao et al. [125] have presented a dependence based representation called the system dependence net (SDN). It helps in representing various dependence relationships in concurrent object-oriented programs like Compositional C++ (CC++) [23] programs. The SDN of a concurrent object-oriented program consists of a collection of dependence graphs each representing a main procedure, a free standing procedure, or

a method in a class of the program. To represent interprocess communications between different methods in a class of a concurrent object-oriented program, a new type of program dependence arc named as external communication dependence arc is introduced into the SDN. Based on the SDN, they have applied a two-phase algorithm to compute static slices of concurrent object-oriented programs. The SDN is actually an extension of the SDG of Larson and Harrold [74] and therefore can be used to represent many object-oriented features in a CC++ program. To handle concurrency issues in CC++, they have used an approach proposed by Cheng [30] which was originally used for representing concurrent procedural programs with a single procedure each. However, their approach [125], when applied to concurrent Java programs suffers from some problems due to the fact that the concurrency models of CC++ and Java are essentially different. While Java supports monitors and some low level thread synchronization primitives, CC++ uses a single assignment variable mechanism to realize thread synchronization. This difference leads to different sets of concurrency constructs in both languages, and therefore requires different techniques to handle concurrency issues in computing slices.

Zhao [123] has also presented a dependence-based representation called the multi-threaded dependence graph (MDG) to represent concurrent Java programs. The MDG is composed of a collection of thread dependence graphs (TDG) each representing a single thread in the program, and some special kinds of dependence arcs to represent thread interactions between different threads. The TDG is used to represent a single thread in a concurrent Java program and is similar to the SDG. The TDG of a thread is an arc-classified diagraph that consists of a number of method dependence graphs each representing a method, and some special kinds of dependence arcs. To represent the synchronization between threads, he [123] has used two special types of dependence arcs in the MDG. He [123] has used synchronization dependence arcs to represent dependence relationships between different threads due to inter-thread synchronization and communication dependence arcs to represent dependence relationships between different threads due to inter-thread communication. Based on the MDG, he has used a two-phase algorithm for computing static slices of concurrent Java programs. He has

not addressed the dynamic slicing aspects of concurrent object-oriented programs.

Cheng [30] introduced a slicing algorithm based on program dependence nets (PDN) for parallel and distributed programs [31]. Both the approaches slice programs by solving a node reachability problem in the graph. A shortcoming of these algorithms is that the resulting slice is not precise since they consider that dependencies between concurrently executed statements are transitive. But, in practice, the dependencies between concurrently executed statements are not transitive due to the presence of synchronization dependence and communication dependence.

To get a more precise slice, Krinke [66, 67] has introduced a new type of dependence called interference dependence, among threads. In Krinke's algorithm, the interference dependence is not transitive. So, the resulting slice is more precise. But, the disadvantage is that, he [66] hasn't considered synchronization issues, while computing slice. However, synchronization is widely used in concurrent programs and in some environments unavoidable. Thus Krinke's algorithm can be used only in some restricted applications. Venkatesh and Hatcliff [96] have developed a robust framework for analysis and context sensitive slicing of concurrent Java programs called Indus. Nanda et al. [90] have extended Krinke's technique to compute static slices of concurrent programs with synchronization. All these approaches don't consider the dynamic slicing aspects.

Mohapatra et al. [87] have also proposed an marking based algorithm for dynamic slicing of concurrent Java programs. They have used concurrent control flow graph (CCFG) and concurrent system dependence graph (CSDG) as the intermediate representations. Based on the CSDG, they have proposed a marking based dynamic slicing (MBDS) algorithm for concurrent Java programs. The MBDS algorithm is based on marking and unmarking the edges of the CSDG as and when the dependencies arise and cease during run-time. MBDS algorithm permanently marks the control dependence edges as control dependencies do not change during program execution. The algorithm considers all the data dependence edges, synchronization dependence edges and communication dependence edges for marking and unmarking during run-time. During execution of the program P, MBDS algorithm marks an edge of the CSDG when

its associated dependence exists, and unmarks when its associated dependence ceases to exist. After each statement u is executed, MBDS algorithm unmarks all incoming marked dependence edges excluding the control dependence edges, associated with the object obj, corresponding to the previous execution of the statement u. Then, the algorithm marks the dependence edges corresponding to the present execution of the statement u.

3.3 Slicing of Distributed Programs

Slicing of object-oriented programs, has been investigated by many researchers. But there is scarcity of resources, that are based on slicing of distributed object-oriented programs [88]. Even the mostly available papers are based on the slicing of distributed procedural programs. Below, we are giving a brief literature survey of slicing of distributed object-oriented programs.

Korel and Ferguson [63] have proposed a dynamic slicing algorithm for Ada programs. According to their approach, each process generates a complete execution trace. Having done the dependence analysis, slices are computed from the trace file. The disadvantage of this approach is that, the entire process was inefficient because of the use of trace file.

Duesterwald et al. [37] have presented a parallel algorithm for computing dynamic slice of procedural distributed programs. They have proposed a new graph called *dynamic dependence graph* (DDG) as the intermediate representation. By solving it as a graph reachability problem, slices are computed from the DDG. They used both static and dynamic information to compute slice. The disadvantage of this algorithm is that, it can't be applied to programs which send and receive messages asynchronously.

Li et al. [76] have given the notion of predicate based dynamic slicing algorithm. Their [76] algorithms are based on partially ordered multi-set (POMSET) model. Unlike traditional slicing criteria, a predicate focuses on those parts of the program, which may influence it. Their proposed algorithm was able to handle message passing system. But they [76] haven't considered object-oriented features.

Mohapatra et al. [88] have proposed a marking based dynamic slicing technique

for computing dynamic slice of a distributed object-oriented program. They [88] have proposed a new graph called distributed program dependence (DPDG) as the intermediate program dependence representation. Their proposed algorithm also known as distributed dynamic slicing (DDS) algorithm works by marking the edges of DPDG, when dependencies arise and unmarking the edges when dependencies cease to exist. The advantage of the algorithm is that it is able to compute precise dynamic slice. But it takes considerable time to compute dynamic slice as the process of marking and unmarking consumes significant amount of time and makes the entire process slower.

3.4 Summary

In this chapter, we have briefly reviewed some work on slicing of object-oriented programs relevant to our research. We have discussed the work on dynamic slicing of object-oriented programs. We have also presented literature study on slicing of concurrent and distributed object-oriented programs. We have discussed the relevant work on slicing of concurrent object-oriented programs. Finally, we have briefly reviewed the available work on slicing of distributed object-oriented programs.

Chapter 4

Dynamic Slicing of Object-Oriented Programs

Efficiency is a key issue in any dynamic slicing technique since the results are normally used in different software engineering activities like program testing, software maintenance, debugging etc. Efficiency is a matter of concern in slicing of object-oriented programs, since object-oriented programs are usually large in size. Hence, if the slicing process is slower, then it would be less useful for object-oriented software development. With this motivation, in this Chapter, we propose a dynamic slicing algorithm, which is more time efficient than the existing dynamic slicing techniques. We have named our proposed algorithm as *Contradictory Graph Coloring Algorithm* (CGCA). We first statically construct the system dependence graph (SDG) as the intermediate representation of the object-oriented program under consideration. Then we use graph coloring technique on the SDG to compute the dynamic slice of the object-oriented programs.

We first discuss the basic concepts. Then, a brief description about the application of graph coloring to program slicing is given. Then, the contradictory graph coloring algorithm is presented. Finally, it is compared with the existing algorithms.

4.1 Basic Concepts

Let *var* be a variable in a program P. In the program P, several statements may define the variable *var*. Let u be a statement that uses the value of the variable *var*. In the PDG G_P of the program P, the node representing the statement u, will have an incoming data

dependence edge corresponding from each of the nodes representing the statements where the variable *var* is defined. Consider the example program given in Fig. 4.1 and

```
1.    main( ) {
      int  m, a, b, i, x, y, z;
2.       cin >> m;
3.       a = 0;
4.       i = 1;
5.       b = 2;
6.       while (i <= m) {
7.             cin >> x;
8.             if (x <= 0)
9.                 y = x + 5;
                  else
10.                y = x − 5;
11.            z = y + 4;
12.            if (z  > 0)
13.                a = a + z;
                   else
14.                b = a + 5;
15.            i = i + 1;
                   }
16.      cout << a;
17.      cout << b;
          }
```

Figure 4.1: An example program

its PDG in Fig. 4.2. In the example program, the statements 9 and 10 define the same variable y. In the PDG of the example program, node 11 has two incoming edges (9, 11) and (10,11) corresponding to the definitions of the variable y. Any iteration of the while loop executes exactly one of the statements 9 and 10, and skips the other. Each of the statements 9 and 10 defines the variable y afresh without using it's previous value directly or indirectly. Therefore, to get a precise dynamic slice for the slicing criterion $< 11, z >$ in any iteration of the while loop, we should consider only the edge (9, 11) or (10, 11) depending on whether the statement 9 or the statement 10 is executed in that iteration.

This observation suggests that by coloring the appropriate nodes of the PDG at run-

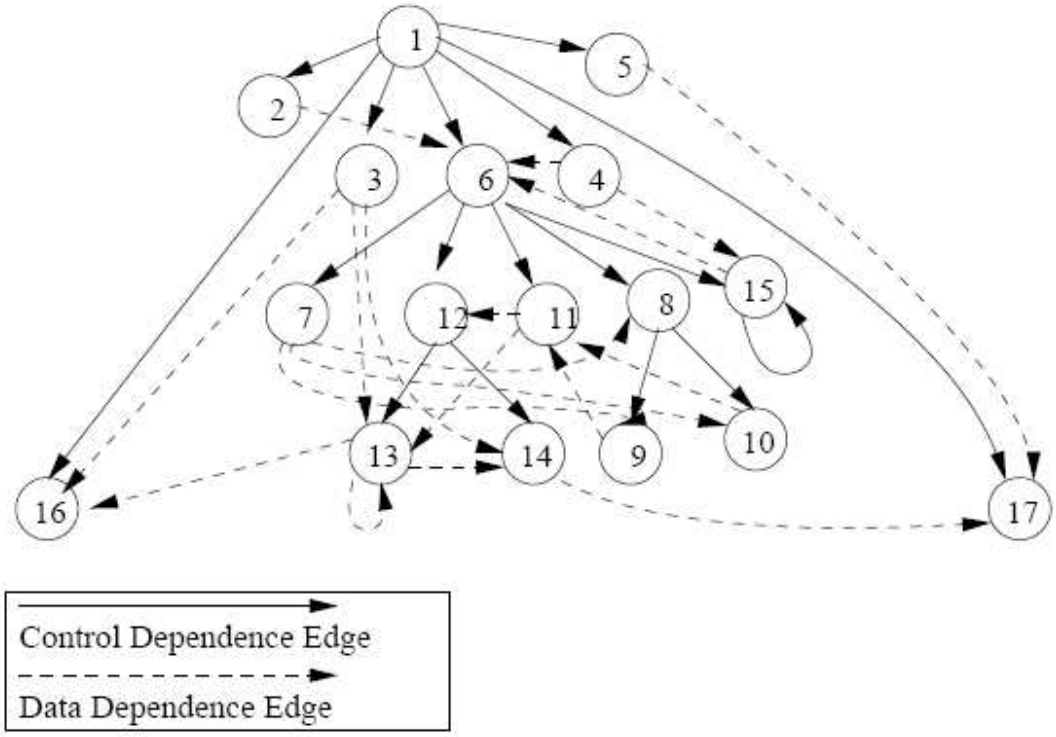

Figure 4.2: The PDG of the program shown in Figure 4.1

time and by doing the dependence analysis, slices can be captured without creating any new node unlike the DODG based algorithm of Zhao [121]. With this motivation, we explore the possibility of development of an efficient dynamic slicing algorithm, and propose an algorithm which is more time efficient than the existing algorithms.

4.2 Our Approach: A Graph Coloring Interpretation to Program Slicing

Graph coloring [35] is a technique which is used for coloring the nodes (edges) so that, no two nodes (edges) sharing the same edge (node) will have the same color. This is called the vertex (edge) coloring problem. Graph coloring has seen wide range of applications such as: *estimation of sparse jacobians*, *job scheduling*, *register allocation* etc.

Hence, according to the definition, the coloring of a graph $G = (V, E)$ is a mapping $c : V \rightarrow S$, where S is a finite set of colors, such that if $(V, W) \in E$, then $c(V) \neq c(W)$. The minimum number of colors required to color the graph is called chromatic number represented as $\chi(G)$. The graph coloring scheme is broadly divided into two types:

- Strong graph coloring
- Weak graph coloring

More literature about graph coloring can be found in [35].

As previously discussed, program slicing is a decomposition technique in which, we extract those statements from the parent program, which may affect the slicing criterion directly or indirectly. In order to compute the slice efficiently, the program is represented as the dependence graph, from which we compute the slice. Mohapatra et al. [86] have developed an marking based dynamic slicing technique for object-oriented programs. The algorithm works by marking those edges for which dependence exists and unmark the edges when the dependence ceases to exist. From their algorithm, we are motivated to incorporate graph coloring scheme in program slicing. Our algorithm is based on the following assumptions:

1. We remove the restriction that "no two vertices sharing the same edge will have the same color".
2. The chromatic number of the graph is taken as one i.e. $\chi(G)$=1.

The principal reason behind contradicting the constraints of the traditional graph coloring algorithm is that the statements corresponding to two vertices sharing the same edge can be present in the slice. Hence, instead of using multiple color, only one color is sufficient to color the vertices. Even our algorithm doesn't fall under the category of weak graph coloring technique; hence we name it as *Contradictory Graph Coloring Algorithm*.

4.2.1 Extending State Restriction to Handle Dependence

Binkley et al. [20] have given a universal view of state restriction as follows:

[State restriction, (↾)]: If σ be the state and V be the set of variables,then $\sigma \upharpoonright V$ restricts σ so that it is defined only for variables of V

$$(\sigma \upharpoonright V) = \begin{cases} \sigma x & \text{if } x \in V \\ \perp & \text{otherwise} \end{cases} \tag{4.1}$$

Since, we have to compute the dynamic slice of an object-oriented program, we have to take into consideration the issues that may arise due to control dependence (cd) or data dependence (dd). As per our requirements, we have modified the definition of state restriction.

In order to capture the proper state restriction, we modify the original definition [20] as below:

"If 'σ' be the given state and V be the set of variables then 'σ' is state restricted by V when there exists either control dependence or data dependence."

In program projection theory notation, it can be written as:

$$(\sigma \upharpoonright V)_{cd||dd} = \begin{cases} \sigma x & \text{if } x \in V \\ \perp & \text{otherwise} \end{cases} \tag{4.2}$$

4.3 Slicing of Object-Oriented Program Using CGCA

In Fig. 4.3, we have shown the example program which is written in C++ and performs a simple mathematical task of addition and increment. The *system dependence graph* (SDG) of the example program of Fig. 4.3 as the intermediate representation for the candidate program as shown in Fig. 4.4. More about SDG can be found in [56].

We now briefly describe our CGC Algorithm. The system dependence graph (SDG) is constructed statically once. The algorithm works by coloring the nodes of the SDG. A node is colored, when it is found to state restrict the slicing criterion depending upon the existence of dependence. To handle method calls, when a statement invokes a method, we color the corresponding called node. Simultaneously, we color the corresponding parameter nodes representing the formal parameter vertices and actual parameter vertices. Now we present our CGC Algorithm to compute dynamic slice of object-oriented programs in pseudo code.

```
Class Task {
        protected:
                int a, b, c;
                int sum, i;
        public:
2               int add (int a, int b) {
3                       c = a + b;
                        return (c);
                }
4               int incr (int a) {
5                       a = a + 1;
                        return (a);
                }
6               void fun ( ) {
7                       sum=0;
8                       i = 1;
9                       while (i < 11) {
10                              sum=add (sum, i);
11                              i = incr (i);
                        }
12                      cout<< "SUM="<<sum;
}
main ( ) {
        Task ob;
        clrscr ( );
1       ob.fun ( );
}
```

Figure 4.3: A simple C++ program doing arithmetic operations

Contradictory graph coloring algorithm for dynamic slicing of object-oriented programs

Input: Set of nodes of SDG ($n_1, n_2, n_3, \ldots, n_m$), n_c being the node representing the slicing criterion , current state σ of slicing criterion, set of variables V ($x_1, x_2, \ldots, x_j$).

Output: Colored nodes showing the slice on the SDG.

1. Draw the SDG statically once.

2. Set the chromatic number χ(SDG)=1 as we are going to use one color to compute the slice.

3. Traverse backward the SDG.

while(*traversedNode* ≤ m)

{

if ((σ ↾ x_j)$_{cd}$ for n_r) , **then**

color the node n_r reached from n_c during traversal

else if ((σ ↾ x_j)$_{dd}$ for all $x_j \in V$) , **then**

{

if the reached node n_r is already colored, **then**

go to Label1

else

color the reached node n_r.

Label1: *traversedNode* = +1;

}

}

4. Look up the nodes which are colored during the graph traversal representing the slice.

4.3.1 Working of The Algorithm

We explain the working of the algorithm with the help of the example program given in Fig. 4.3. For the given program, we first construct the SDG. Then during the execution of the program, we update our SDG. Here, we update the graph by coloring those nodes which are able to state restrict the slicing criterion either by control dependence or by data dependence. For the given argument values, we will apply our proposed contradictory graph coloring algorithm to compute dynamic slice. For the example, we illustrate the working of our algorithm for computing the dynamic slice with respect to the slicing criterion < 10,*sum* > for the input value of i=4.

The SDG consists of several elements where the circles or nodes represent the statements of the program and the ellipses represent the parameters for the various

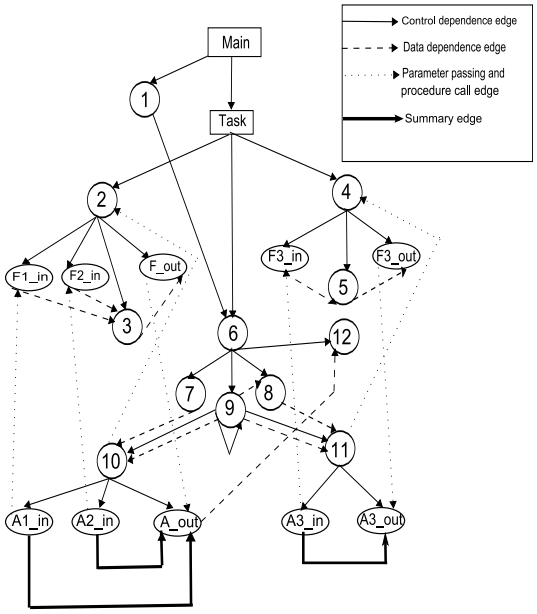

Figure 4.4: The SDG of the program shown in Fig. 4.3

functions. We traverse the nodes of the SDG backward in order to compute the backward dynamic slice of the given object-oriented program. In the SDG, node '10' represent the slicing point. We assume that the node representing the slicing point, is the first node to be traversed and the process is continued till every node is traversed. We give an example for the traversal of node '9'. We find that node '10' is control dependent on the reaching node '9'. Hence, we color the node '9'. Similarly, during every iteration of the while loop, a new node is traversed and depending upon the existence of the type of dependence and state restriction, the reaching node is colored. One important point to be noted in the above discussed algorithm is that, if a node is already colored, it needn't be re-colored again, even if it is found to state restrict the slicing criterion. The consequence is that, it helps in speeding up the process of computing slice. After the termination of the while loop, we just need to look up for those nodes which are colored during the process. In Fig. 4.5, we show the updated SDG with colored nodes representing the slice of the example program shown in Fig. 4.3 with respect to the slicing criterion $< 10, sum >$.

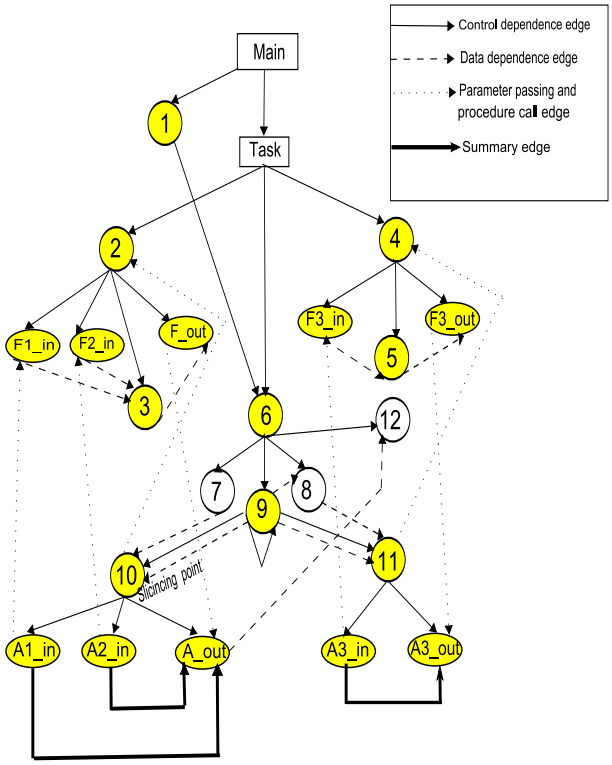

Figure 4.5: The updated SDG of the program shown in Fig. 4.3 with colored nodes representing the slice

4.4 Implementation Results

The proposed *contradictory graph coloring algorithm* is implemented in Java. We have also implemented the EMDS algorithm proposed by Mohapatra et al. [86]. We have used the complier writing tool ANTLR (Another Tool for Language Recognition) [59, 83] for lexical analysis, parser and semantic analysis. ANTLR allows one to define grammar in EBNF (Extended Backus Naur Form) notations. It also lets one to implement ANTLR defined grammar by generating lexers and parsers in Java. The program to be sliced is given as an input to the ANTLR. The SDG of the input program is constructed by taking input from the parser and semantic analyzer components. Then, the dynamic slice is computed by running the CGC algorithm using the SDG. The results of our implementation are summarized in Table 4.1. We have compared our algorithm with EMDS algorithm. Fig. 4.6 shows the graphical representation of the comparison between CGCA and EMDS algorithm.

In edge marking dynamic slicing (EMDS), the ESDG is updated by marking and

Table 4.1: Comparison of average run-time between EMDS and CGCA

Sl. No.	Program Size (No. of stmt.)	Average Runtime (in Sec.)	
		EMDS	CGCA
1.	12	0.06	0.04
2.	79	0.18	0.14
3.	125	0.21	0.16
4.	180	0.25	0.19
5.	253	0.32	0.25
6.	327	0.39	0.31
7.	415	0.47	0.38
8.	513	0.58	0.47
9.	605	0.70	0.59

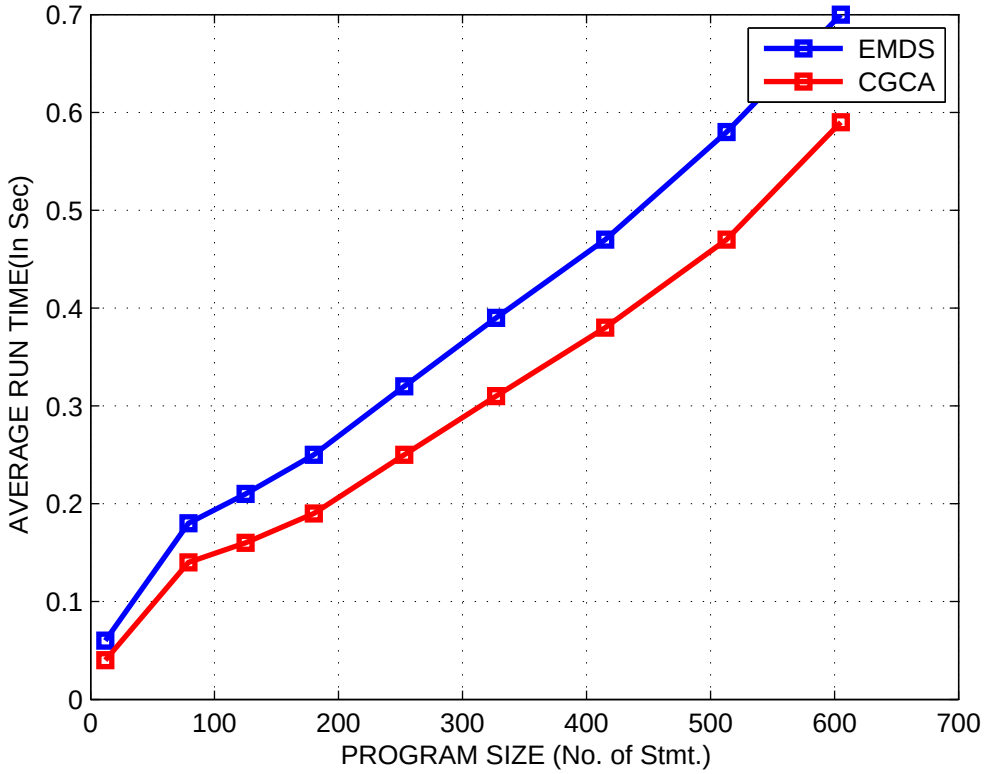

Figure 4.6: Comparison of average run-time between CGCA and EMDS algorithm

unmarking of the edges. When the dependence exists the, the edges are marked and when the dependence ceases to exist, the edges are unmarked. Hence the process of marking and umarking consumes considerable time in updating the graph.

On the other hand, CGCA works by coloring the nodes of the SDG. Apart from dependence analysis, we have taken into consideration the state of the slicing criterion. Our algorithm colors a node if and only if it is found to state restrict the slicing criterion. Hence unlike the EMDS algorithm, time is consumed only to color the nodes

representing the slice. So, our algorithm takes less time than that of EMDS algorithm.

4.5 Comparison With Related Work

Larsen and Harrold [74] were the first to consider the slicing of object-oriented programs. Their main contribution was towards the representation of dependence graph for single class, interacting classes and complete object-oriented programs. So according to their approach, it was possible to compute the slice of complete as well as incomplete systems. They had developed a graph called *class dependence graph* (ClDG) for representing an individual class. It was also found to be helpful in successful representation of different object-oriented features like polymorphism, inheritance etc. They have computed static slice of object-oriented programs using two-pass algorithm [18]. Their algorithm can not be used for computing dynamic slices.

Zhao [121] developed a different intermediate representation for programs called *dynamic object-oriented dependence graph* (DODG). He was the first to compute the dynamic slice of object-oriented program. His proposed DODG was basically an arc-specified digraph to represent various dynamic dependences among the statements during the execution. The process required to create new nodes to compute dynamic slice. But in our proposed algorithm, no new node is required to be created. By applying the vertex coloring technique, dynamic slice is computed. So our algorithm is more faster than Zhao's technique.

Ohata et al. [93] have proposed a unique program slicing method for object-oriented programs to evaluate its effectiveness with Java programs. They [93] have adopted an intermediate slicing method, that lies between static and dynamic slicing. They have named the new technique as *Dependence-Cache* (DC) slicing to object-oriented programs. DC slicing method uses dynamic data dependence analysis and static control dependence analysis. The DC slice is computed in four phases as discussed earlier. They have proposed an algorithm, named *Object-Oriented Dynamic Cache algorithm* to compute DC slice of object-oriented programs. The size of OODC slice is claimed to be 20% - 70% as large as that of static slice, hence more precise than static slice. But the

DC slices are found to be larger than dynamic slice. Since the entire process works in four phases, it is very slow compared to our CGC algorithm. Our proposed algorithm works iteratively to color the nodes of the SDG to compute dynamic slice.

Xu et al. [114] computed dynamic slices of object-oriented programs using the object program dependence graph (OPDG). In order to implement their method, Xu et al. [114] inserted breakpoints to some statements to record the execution information. For inserting breakpoints to the statements some additional space and time is required. But in our CGC algorithm, we do not insert any breakpoint to the statements. Again, they have used trace files to store the execution history, where as our CGC algorithm does not use any trace files. Thus, it is clear that our CGC algorithm is more efficient than the algorithm proposed by Xu et al. [114].

Song and Huynh [101] have proposed a slicing technique to compute forward dynamic slice of object-oriented programs. They have proposed an algorithm to slice object-oriented programs with respect to a slicing criterion that recursively decomposes constructors and member functions. According to their approach, they decompose the instance variables of each object beginning by slicing its constructors. The resulting slices are reused while the instance variables remain untouched. They have also developed a dynamic object relationship diagram (DORD) that shows the interaction of objects with respect to specified variables. The dynamic slices of all executed statements are obtained only when the last statement of the program is executed. Our approach does not require to wait, till the program execution ends to get the dynamic slice. So our approach is more efficient than the algorithm proposed by Song and Huynh [101].

Zhang and Gupta [119] have proposed the LP algorithm, where the dynamic dependence graph was constructed on-demand in response to dynamic slicing requests from the execution trace that was saved on disk. While their developed approach was able to greatly reduce the size of dynamic dependence graph held in memory, the on-demand construction of the dynamic dependence graph was quite slow, since it used to require repeated traversals of the trace stored on disk. To enable faster traversal of the trace, they had augmented the trace with summary information which allowed us

to skip irrelevant parts of the trace during traversal. However, the overall process was slow because of the use of trace file.

Mohapatra [86] has proposed a marking based algorithm for computing the dynamic slice of object-oriented programs. He has proposed an algorithm which is known as *edge marking dynamic slicing* (EMDS). His algorithm is based on marking and unmarking of edges (or nodes) of the ESDG as and when dependences arise and cease to exist. The disadvantage of the algorithm is that, it consumes more time in marking and unmarking of nodes of the ESDG. Our algorithm works by coloring the nodes of the SDG. When the slicing criterion is found to be state restricted, then only a node is colored. So our algorithm is more time efficient, than the EMDS algorithm.

Binkley et al. [21] have developed an algorithm for variable substitution that allows the dependence due to a particular global variable to be ignored, which is used to assess and measure the effects of the global variable on dependence. They have presented quantitative results that assess the effect on dependence of 849 global variables in 21 programs and revealed that more than half the programs considered have individual global variables that have a significant impact on overall program dependence. They have also presented qualitative results of the effect these high dependence globals have on large dependence clusters. In some cases, a single global was found to be the sole cause of a cluster, establishing evidence for a link between the use of globals and the presence of large dependence clusters. But they haven't considered object-oriented features in their proposed framework.

4.6 Conclusion

In this chapter, we have proposed a novel dynamic slicing algorithm for object-oriented programs. We have named our algorithm *contradictory graph coloring* (CGC) algorithm. Our algorithm uses SDG as the intermediate representation. CGC algorithm is based on coloring of the nodes of the SDG, when the dependencies exist and the slicing criterion is state restricted during run-time. The advantage of our algorithm is that, it doesn't require any trace file to be maintained. So, the expensive file I/O operation is

not required during the entire process. Also, it doesn't take extra time for marking and unmarking the edges. Although, we have presented our dynamic slicing technique for C++ programs, it can easily be adapted to programs written in other object-oriented languages such as Java. The algorithms presented in this chapter cannot handle concurrency issues in object-oriented programs. In the next chapter we extend our framework to consider concurrency issues in object-oriented programs.

Chapter 5

Dynamic Slicing of Concurrent Object-Oriented Programs

Now-a-days, many of the object-oriented softwares, available in the market are concurrent in nature. It is more difficult to comprehend and test an concurrent object-oriented program, than an ordinary sequential program. The non-deterministic nature of concurrent programs, the lack of global states, unsynchronized interactions among processes, multiple threads of control and a dynamically varying number of processes are some reasons for this difficulty. During the object-oriented software development, a huge amount of resource is spent for testing and maintenance of the software. At this point, program slicing can be applied effectively to lessen the burden. Moreover, research work available, to address the problem of dynamic slicing of concurrent object-oriented programs is very scarce. The objective of this chapter is to develop an efficient algorithm to compute the dynamic slice of a concurrent object-oriented program.

The intention of any slicing technique is efficiency as the results are widely used in many day-to-day software activities like maintenance, testing etc. Efficiency is a matter of concern for slicing of concurrent object-oriented programs, since their sizes are typically very large. As the program size grows gradually, the response time also increases by several hundreds of seconds.

With this motivation, in this chapter we propose a new dynamic slicing algorithm for computing slices of concurrent Java programs. Only the concurrency issues are addressed here, traditional object-oriented features are not discussed in this chapter. However, the traditional object-oriented features in program slicing can be found

in [56]. Our proposed algorithm uses the concurrent system dependence graph, a modified form of the program dependence graph (PDG) as the intermediate representation. The intermediate representation i.e. CSDG is first statically constructed. Then, we apply our algorithm to the CSDG to compute dynamic slices of concurrent object-oriented programs.

Our algorithm is based on coloring of the nodes of CSDG appropriately when the dependencies arise and the state of the slicing criterion is restricted at run-time. But in order to achieve efficiency, we have contradicted the key constraints of the original graph coloring technique. So, the algorithm is named contradictory graph coloring algorithm (CGCA) for concurrent object-oriented programs. Such an approach is more time efficient and also allows to completely eliminate the use of a trace file at run time to record the execution history. In dynamic slicing, it is desirable to eliminate the slow file I/O operations that occurs while accessing a trace file as these make the response times unacceptably large in interactive sessions.

We first present some basic concepts and definitions that will be used in our algorithm. Then, we discuss the about concurrent system dependence graph (CSDG), the intermediate program representation for concurrent object-oriented programs. Next, we present our contradictory graph coloring algorithm (CGCA) for computing dynamic slice of concurrent object-oriented programs. Then, we discuss about the working of the algorithm.

5.1 Concurrency Property of Java

In concurrent programming [7, 13], there are two basic units of execution: processes and threads. In the Java programming language, concurrent programming [8] is mostly concerned with threads. However, processes are also important. A computer system normally has many active processes and threads [9]. This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called time slicing.

It's becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores. This greatly enhances a system's capacity for concurrent execution of processes and threads. But concurrency is possible even on simple systems, without multiple processors or execution cores.

Difference between processes and threads: A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources. In particular, each process has its own memory space. Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support Inter Process Communication (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.

Most implementations of the Java Virtual Machine (JVM) run as a single process. A Java application can create additional processes using a Process Builder Object. Multiprocess applications are beyond the scope of this Chapter.

Threads are sometimes called lightweight processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process. Threads exist within a process. Every process has at least one thread. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread or several, if you count "system" threads that do things like memory management and signal handling. But, from the application programmer's point of view, we start with just one thread, called the main thread. This thread has the ability to create additional threads.

A thread is a program's path of execution. Most programs written today run as a single thread, causing problems when multiple events or actions need to occur at the same time. Let's say, for example, a program is not capable of drawing pictures while reading keystrokes. The program must give its full attention to the keyboard input

lacking the ability to handle more than one event at a time. The ideal solution to this problem is the seamless execution of two or more sections of a program at the same time. Threads allows us to do this.

Multithreaded applications deliver their potent power by running many threads concurrently within a single program. From a logical point of view, multithreading means multiple lines of a single program can be executed at the same time, however, it is not the same as starting a program twice and saying that there are multiple lines of a program being executed at the same time. In this case, the operating system treats the programs as two separate and distinct processes. Under Unix, forking a process creates a child process with a different address space for both code and data. However, fork() creates a lot of overhead for the operating system, making it a very CPU-intensive operation. By starting a thread instead, an efficient path of execution is created while still sharing the original data area from the parent. The idea of sharing the data area is very beneficial, but brings up some areas of concern that we'll discuss later.

Java has been graciously designed with two ways of creating threads: implementing an interface and extending a class. Extending a class is the way Java inherits methods and variables from a parent class. In this case, one can only extend or inherit from a single parent class. This limitation within Java can be overcome by implementing interfaces, which is the most common way to create threads.

Java supports concurrent programming using threads. A thread is a single sequential flow of control within a program. A thread is similar to a sequential program in the sense that each thread also has a beginning, an execution sequence and an end, and at any given instant of time during the run of the thread, there is a single point of execution. However, a thread itself is not a program; it cannot run on its own. Rather, it runs within a program. A program that contains multiple threads is called a multi-threaded program. Java supports shared memory-based communication among threads. Objects shared by two or more threads are called condition variables, and the access to them must be synchronized for the proper working of the system. The Java language and run-time system support thread synchronization through the use of monitors. A monitor is associated with a specific data item and functions to lock that

data. When a thread holds the monitor for some data item, other threads are locked out and cannot inspect or modify the data. The code segments within a program that access the same data from within separate threads are known as critical sections. In Java programs, critical sections may be marked with the keyword synchronized. To support synchronization among different threads, Java provides some methods such as *wait( )*, *notify( )* and *notifyall( )* etc.

5.2 Intermediate Representation for Concurrent Programs

In order to compute the slice of concurrent object-oriented programs efficiently, we have chosen the *Concurrent System Dependence Graph* (CSDG) as the intermediate program representation as described by Mohapatra et al. [87].

Construction of CCFG and CSDG

A thread is a single sequential flow of control in Java. Hence each thread can be represented by control flow graphs (CFG). Since synchronization is a matter of concern, the control flow among the various threads are not independent. Hence, it is necessary to incorporate required changes in the traditional CFG to address the issue of synchronization. The modified graph is known as Concurrent Control Flow Graph (CCFG).

CCFG represents the higher level abstraction of the CSDG. CSDG is constructed by adding the synchronization dependence edges and communication dependence edges to capture the concurrency aspects of object-oriented programs. CSDG of a concurrent object-oriented program defines the program dependencies, that can be determined statically as well as the dependencies that may exist at run time. Control dependence can be determined statically at the compilation time as it doesn't change during run time. The dependencies which do arise at run time are data dependence, synchronization dependence and communication dependence. We use different types of edges to denote these different dependencies. These edges are:

- Control dependence edge: - If node y is control dependent on node x, then the

edge $E(x, y)$ is called a control dependence edge. In Fig. 5.2 the edges (14, 15) and (15, 17) represent the communication dependence edges.

- Data dependence edge: - If node y is data dependent on node x, then the edge $E(x, y)$ is called a data dependence edge. In Fig. 5.2 the edges (18, 20) and (16, 9) represent the data dependence edges.

- Synchronization dependence edge: - If node y is synchronization dependent on node x, then the edge $E(x, y)$ is called a synchronization dependence edge. In Fig. 5.2 the edges (3, 8) and (10, 5) represent the synchronization dependence edges.

- Communication dependence edge: - If node y is communication dependent on node x, then the edge $E(x, y)$ is called a communication dependence edge. In Fig. 5.2 the edges (9, 6) and (12, 4) and (13, 6) represent the communication dependence edges.

  The CSDG of a concurrent object-oriented program P can be constructed through the following steps:

  Step 1: Construct the CCFG for P.

  Step 2: Recursively delete the nodes that cannot be reached from the entry node of P's CCFG and the synchronization edges that cannot be triggered.

  Step 3: Construct a system dependence graph (SDG) by using this modified CCFG as given in [58].

  Step 4: Add synchronization dependence edges and communication dependence edges to PDG resulting after step 3, in order to get the CSDG.

```
class Thread1 extends Thread {
  private SyncObject O;
  private CompObject C;
  void Thread1 (SyncObject O, CompObject a1,
   CompObject a2, CompObject a3)
  {
   this.O = O;
   this.a1 = a1;
   this.a2 = a2;
   this.a3 = a3;
  }
1  public void run ( ) {
2   a2.mul (a1, a2);
3   O.Snotify ( );
4   a1.mul (a1, a3);
5   O. Swait ( );
6   a1.mul (a2, a2);
  }
}
class Thread2 extends Thread {
  private SyncObject O;
  private CompObject C;
  void Thread1 (SyncObject O, CompObject a1,
   CompObject a2, CompObject a3)
  {
   this.O = O;
   this.a1 = a1;
   this.a2 = a2;
   this.a3 = a3;
  }
7  public void run ( ) {
8   O. Swait ( );
9   a2.mul (a1, a1);
10   O.Snotify ( );
11   if (a1! = a2)
12    a3. mul (a2, a1);
   else
13    a2.mul (a1, a1);
  }
}

14    Class Example {
15 Public static void main (String arg [ ]) {
   CompObject a1, a2, a3;
   SyncObject o1;
   o1.reset ( );
16   a1 = new CompObject (Integer.parseInt [0]);
17   a2 = new CompObject (Integer.parseInt [1]);
18   a3 = new CompObject (Integer.parseInt [2]);
19   Thread1 t1 = new Thread (o1, a1, a2, a3);
20   Thread1 t2 = new Thread (o1, a1, a2, a3);
21   t1. Start ( );
22   t2. Start ( );
 }
       }
```

Figure 5.1: A concurrent Java Program

5.3 Contradictory Graph Coloring Algorithm

In this section, a brief description about the proposed Contradictory Graph Coloring Algorithm (CGCA) is given. Then, we represent the pseudo-code of our algorithm.

5.3.1 A Graph Coloring Approach to Program Slicing

Graph coloring is a technique which is used for coloring the nodes (edges) so that, no two nodes (edges) sharing the same edge (node) will have the same color.This is known as vertex coloring problem. And the alternate one is called edge coloring problem. Graph coloring has seen wide range of applications such as: *estimation of sparse jacobians, job scheduling, register allocation* etc.

Hence as per the definition, the coloring of a graph $G = (V, E)$ is a mapping $c : V \rightarrow S$, where S is a finite set of colors, such that if $(V, W) \in E$, then $c(V) \neq c(W)$. The minimum number of colors required to color the graph is called chromatic number represented as $\chi(G)$. More literature on graph coloring can be found in [35].

As previously discussed, program slicing is a decomposition technique in which, we extract those statements from the parent program, which may affect the slicing criterion directly or indirectly. In order to compute the slice efficiently, the program is represented as a dependence graph, from which we compute the slice. Mohapatra et al. [87] have developed an marking based dynamic slicing technique for concurrent object-oriented programs. The algorithm works by marking those edges for which dependence exists and unmark the edges when the dependence ceases to exist. From their algorithm, we are motivated to incorporate graph coloring scheme in program slicing. But here we have to remove two key constraints of the graph coloring technique:

1. We remove the restriction that ”no two vertices sharing the same edge will have the same color”.

2. The chromatic number of the graph is taken as one i.e. $\chi(G)$=1.

The principal reason behind contradicting the constraints of the traditional graph coloring algorithm is that two vertices sharing the same edge can be the slice. Hence, instead of using multiple color, only one color is sufficient to color the vertices. Even our algorithm doesn't fall under the category of weak graph coloring technique; hence we name it as *Contradictory Graph Coloring Algorithm*.

5.3.2 Extending State Restriction To Handle Dependencies

Binkley et al. [20] have given a universal view of state restriction as given in Eq. 2.1. Since, we are going to compute the dynamic slice of a concurrent object-oriented program, we have to take into consideration the issues which may arise due to control dependence (cd), data dependence (dd), synchronization dependence (sd) or communication dependence (cmd). Hence as per our requirements we have specialized the definition of state restriction.

In order to capture the proper state restriction, we modify the original definition as below:

$$(\sigma \upharpoonright V)_{cd||dd||sd||cmd} = \begin{cases} \sigma x & \text{if } x \in V \\ \perp & \text{otherwise} \end{cases} \tag{5.1}$$

5.4 Slicing of Concurrent Object-Oriented Programs Using CSDG

In Fig. 5.1, we have shown an example Java program taken from [87] for which we are going to compute the slice using our proposed new algorithm and the CSDG.

First, we provide a brief overview of our proposed contradictory graph coloring algorithm for dynamic slicing of concurrent object-oriented programs. The concurrent system dependence graph (CSDG) is constructed statically once by using the CCFG. We consider all the data dependence edges, synchronization dependence edges and communication dependence edges for coloring during run-time. During the execution of the program, the type of dependence that exists is found. If the statement being executed is found to state restrict the slicing criterion, then the node representing that statement is colored.

We now present our CGC Algorithm for concurrent object-oriented programs in the form of pseudo-code.

Contradictory graph coloring algorithm for dynamic slicing of concurrent object-oriented programs

Input: The set of nodes ($n_1, n_2, n_3, \ldots, n_m$) of CSDG, n_c being the node representing the slicing criterion, current state σ of slicing criterion, set of variables V ($x_1, x_2, \ldots, x_j$)
output: Colored nodes showing the slice on the CSDG.

1. Draw the CSDG statically once.
2. Set the chromatic number χ(CSDG)=1 as we are going to use one color to compute the slice.
3. Traverse backward the graph CSDG.

```
while(traverseNode ≤ m)
{
    n_i = traverseNode ;
    if ( ( σ ↑ x_j)_cd for all x_j ∈ V)  , then
        color the node reached n_r from n_c during traversal
    else if ( ( σ ↑ x_j)_dd for all x_j ∈ V)  , then
    {
        if the node reached n_r is already colored, then
            go to label1
        else
            color  the node reached n_r.
    }
Label1:else if( (σ ↑ x_j)_sd||cmd for all x_j ∈ V),  then
    {
        if the node reached n_r is already colored, then
        {
            traversedNode=+1;
            continue;
        }
```

```
            else
            {
                color the node n_r reached;
                traversedNode=+1;
            }
        }
}
```

4. Look up the nodes which are colored during the graph traversal representing the slice.

5.5 Working of The Algorithm

We explain the working of the algorithm with the help of the program given in Fig. 5.1. For the given program, we first draw the CSDG as shown in Fig. 5.2. Then during the execution of the program, we update our CSDG. For the given argument values, we will apply our proposed contradictory graph coloring algorithm to compute dynamic slice. Here, we update the graph by coloring those nodes which are state restricted by either control dependence, data dependence or synchronization dependence. In the example, we illustrate the working of the algorithm for computing the dynamic slice with respect to the slicing criterion $< 6, a1 >$.

During the first iteration of the while loop, we traverse the node '6'. Here we assume, that the first node to be traversed should be the node representing the slicing criterion. We find that node '6' is control dependent on the reaching nodes '5' and '1'. Hence, we color the nodes '5' and '1'. Moreover, there is communication dependence between node '6' and '9'. Hence, node '9' is colored. Similarly, during every iteration of the while loop, a new node is traversed and depending upon the existence of the type of dependence and state restriction, the reaching node is colored. One important point to be noted in the above discussed algorithm is that, if a node is already colored, it needn't be re-colored again, even if it is found to state restrict the slicing criterion. The consequence is that, it helps in speeding up the process of computing slice. After the termination of the **while** loop, we just need to look up for those nodes which are

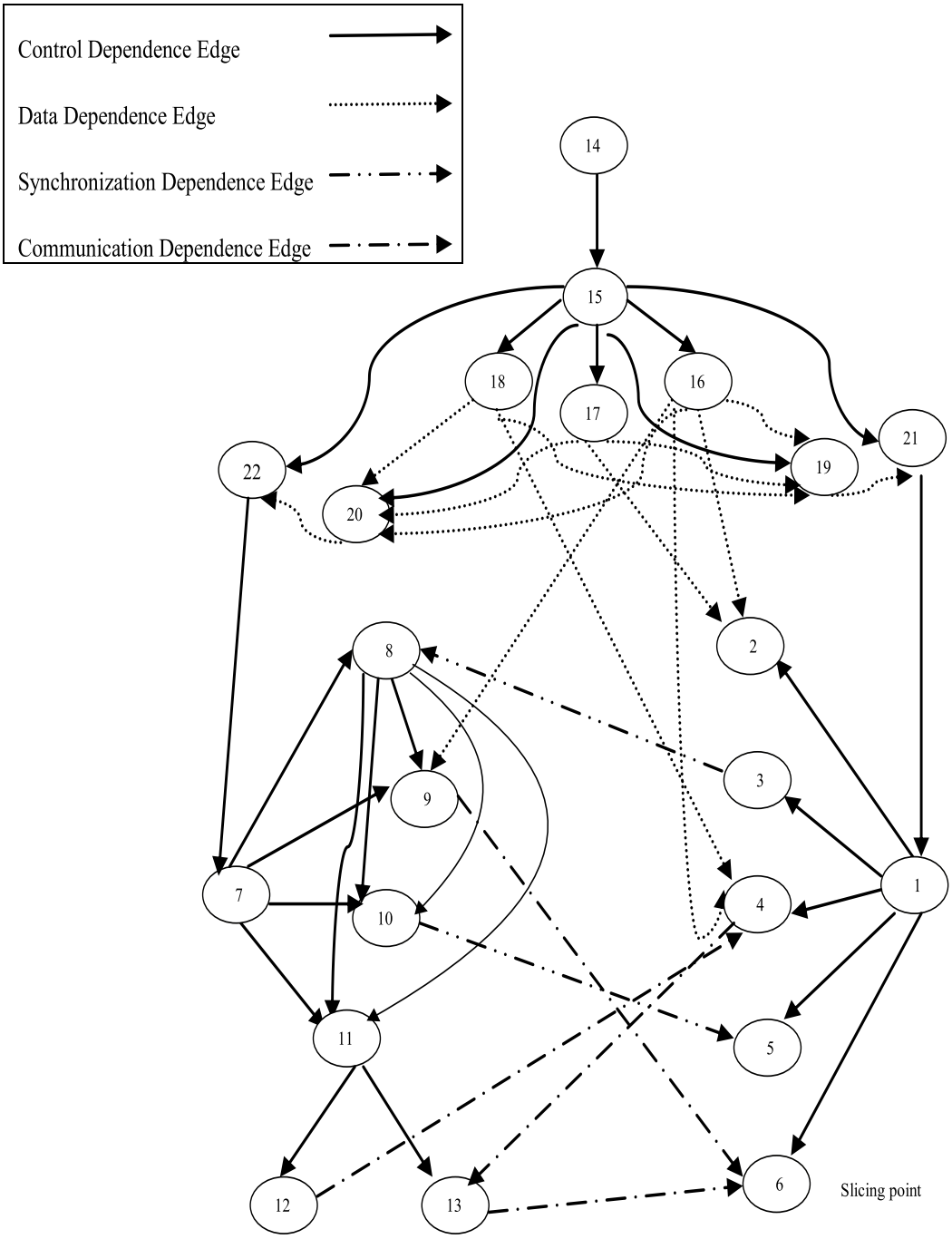

Figure 5.2: The CSDG of the program given in Fig. 5.1

colored during the process. In Fig. 5.3, we show the updated CSDG representing the dynamic slice of the example program shown in Fif. 5.1 with respect to the slicing criterion $< 6, a1 >$.

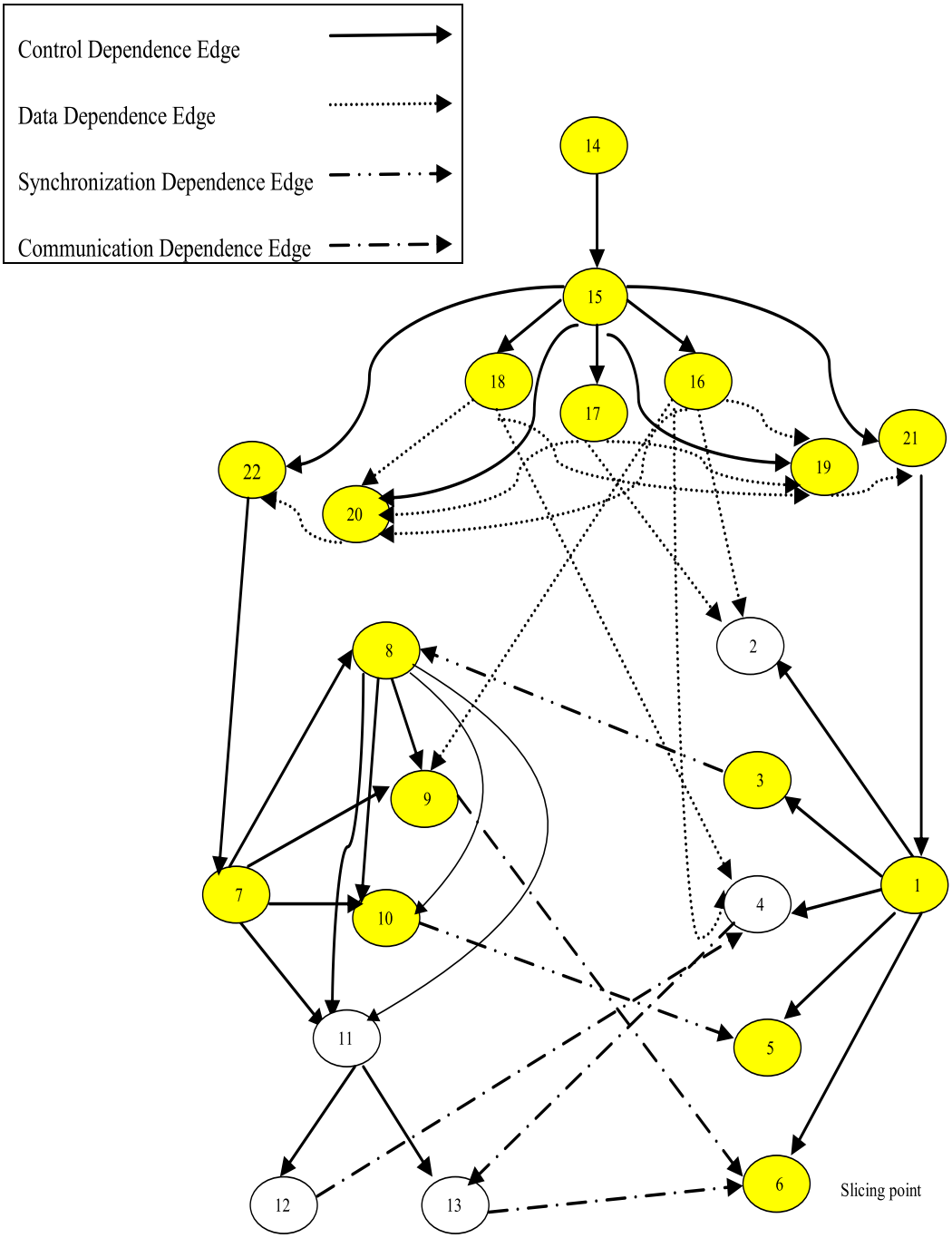

Figure 5.3: The CSDG of the program with colored nodes showing the slices w.r.t. slicing criterion < 6,$a1$ >

5.6 Implementation Results

To study the performance of CGCA, we have simulated it using Java to compute the dynamic slice of concurrent object-oriented programs. Also, we have implemented the MBDS algorithm of Mohapatra et al. [87] for comparison. ANTLR [59], a compiler

designing tool is used for lexical analyzer, parser and semantic analyzer components. It allows one to implement ANTLR defined grammar by generating lexers and parsers in Java. The concurrent object-oriented is given as input to the program written in ANTLR. The CSDG of the input program is constructed by taking input from the parser and semantic analyzer components. Then, applying the proposed CGCA, the dynamic slices are computed. The performance of the algorithm was analyzed carefully. The results of the same are given in the Table 5.1. Fig. 5.4 shows the graphical representation between CGCA and MBDS algorithm.

Table 5.1: Comparison of average run-time between MBDS and CGCA for concurrent object-oriented programs

Sl. No.	**Program Size (No. of stmt.)**	**No. of Threads**	**Average Runtime (in Sec.)**	
			MBDS	CGCA
1.	22	2	0.08	0.06
2.	125	2	0.24	0.18
3.	234	2	0.32	0.25
4.	315	2	0.41	0.32
5.	387	2	0.45	0.35
6.	450	2	0.50	0.39
7.	520	3	0.57	0.45
8.	625	3	0.66	0.54
9.	719	3	0.75	0.62

In marking based dynamic slicing (MBDS), the CSDG is updated by marking and unmarking of the edges. When the dependence exists the, the edges are marked and when the dependence ceases to exist, the edges are unmarked. Hence the process of marking and umarking consumes considerable time in updating the graph.

On the other hand, CGCA works by coloring the nodes of the CSDG. Apart from dependence analysis, we have considered the state of the slicing criterion. Our algorithm colors a node iff it is found to state restrict the slicing criterion. Hence unlike the MBDS algorithm, time is consumed only to color the nodes representing the slice. So, our algorithm takes less time to compute the dynamic slice than that of MBDS algorithm.

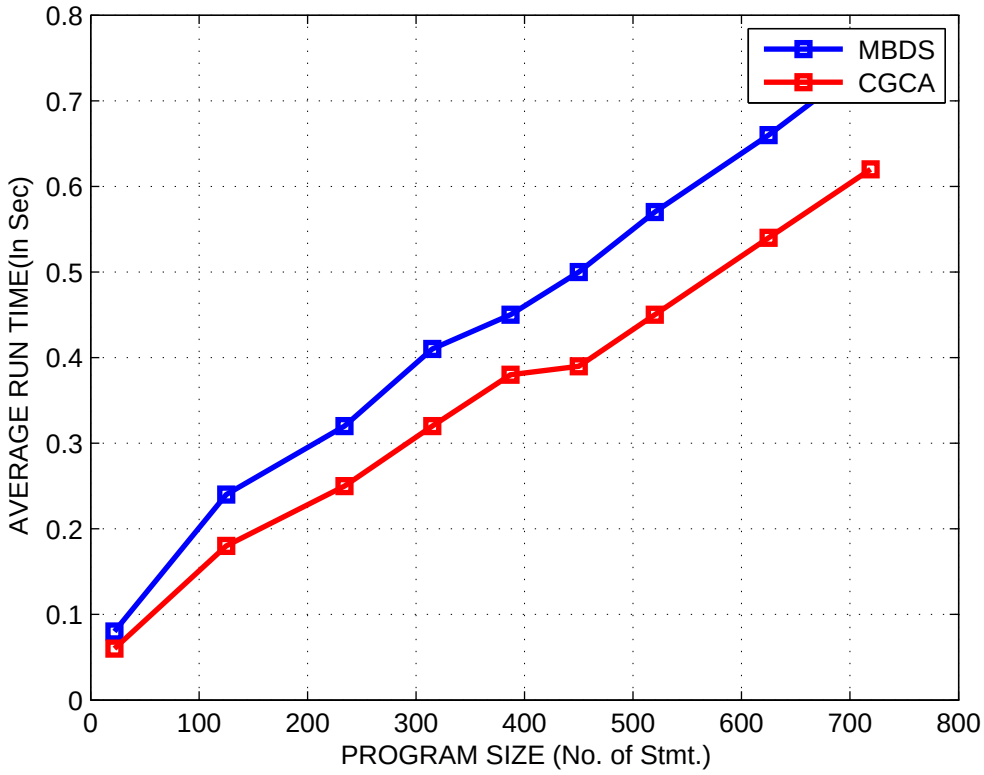

Figure 5.4: Comparison of average run-time between CGCA and MBDS algorithm

5.7 Comparison With Related Work

Cheng [30] considered the problem of slicing concurrent imperative programs. He proposed a program dependence representation, called the Process Dependence Net (PDN) which is the generalization of the PDG to represent program dependences in a concurrent imperative program with single procedure. In addition to the usual control and data dependence arcs as in the PDG, the PDN also contains selection, synchronization, and communication dependence arcs to represent program dependences related to non-deterministic selections, inter-process synchronizations and inter-process communications respectively in the program. Based on the PDN, the problem of slicing concurrent imperative programs can also be simplified to a vertex reachability problem in the net. Their approach deals with computing the static slice of Compositional C++ (CC++) [23]. Whereas, our proposed algorithm deals with computing the dynamic slice of concurrent object-oriented programs.

Chen and Xu [25] developed a novel algorithm to compute static slices of concurrent Java programs. To compute the slices, they used concurrent control flow graph (CCFG) and concurrent program dependence graph (CPDG) as the intermediate representations. In their algorithm, they have considered the fact that the inter-thread data dependencies are not transitive. So, the resulting slice is more precise than that of Zhao [123]. However, they have not considered the dynamic slicing aspects.

Krinke [67] introduced a new type of dependence called interference dependence. In Krinke's algorithm, the interference dependence is not transitive. So, the resulting slices are more precise. But, Krinke has not considered thread synchronization in the algorithm. However, synchronization is widely used in concurrent programs and is normally considered inevitable in concurrent programs. Thus, Krinke's algorithm can be used only in very restricted situations. Nanda and Ramesh [90] have extended Krinke's technique [67] to compute static slices of concurrent programs with synchronization. We have computed the *dynamic* slices of concurrent object-oriented programs, which are smaller than the static slices and hence more useful in different software engineering activities.

Mohapatra et al. [87] have proposed algorithm for dynamic slicing of concurrent Java programs without using trace files. They have used concurrent control flow graph (CCFG) and concurrent system dependence graph (CSDG) as the intermediate representations. Based on the CSDG, they have proposed a marking based dynamic slicing (MBDS) algorithm for concurrent Java programs. The MBDS algorithm is based on marking and unmarking the edges of the CSDG as and when the dependencies arise and cease during run-time. MBDS algorithm permanently marks the control dependence edges as control dependencies do not change during program execution. The algorithm considers all the data dependence edges, synchronization dependence edges and communication dependence edges for marking and unmarking during run-time. The disadvantage of this algorithm is that, it consumes considerable amount of time in marking and unmarking the nodes of the CSDG. But our algorithm computes dynamic slice by iteratively coloring the nodes of the CSDG, which is more faster because time is not wasted in unmarking the nodes of the CSDG.

5.8 Conclusion

In this chapter, we have proposed a novel algorithm for computing dynamic slices of concurrent object-oriented programs. We have named our algorithm as *contradictory graph coloring* (CGC) algorithm. We have considered the Java concurrency model, although it can easily be extended to handle other concurrency models. Our algorithm uses the concurrent system dependence graph (CSDG) as the intermediate representation. CGCA is based on coloring the nodes of the CSDG, when the dependencies exist and the slicing criterion is state restricted during run-time. Our algorithm does not use any trace file to store the execution history. Also, it does not create additional nodes during run-time. This saves the expensive file I/O operations. The algorithm in this chapter is not suitable to be applied to distributed programs running on several nodes connected through a network. In the next chapter, we are extending our framework to compute dynamic slices of distributed object-oriented programs running on several nodes as is common in Client-Server applications.

Chapter 6

Dynamic Slicing of Distributed Object-Oriented Programs

The distributed application development process is the result of a commitment to continuous learning, refinement of experience and improvement of process as new ideas have been developed and new technologies have been employed. It is a collection of experiences and best practices that have been taken from real-world development engagements, providing development teams with access to shared experiences and a proven, repeatable process. The distributed application development process encompasses modern design principles and proven practices to facilitate the development task and provide developers with a blueprint for building robust and correct distributed applications.

Technically, distributed application development is based on a multi-tiered development architecture. In its simplest form, with two tiers, a distributed application is synonymous with a client/server protocol in which you use a set of rules that specify a behavior for two collaborating processes. In a client/server relationship, one process (the client) initiates the interaction by issuing a request to a second process (the server). The server process must await a request from the client and, on receipt of that request, performs a service and returns a response (or result) to the client. The server is capable of handling requests from multiple clients and is responsible for coordinating and synchronizing responses.

Although the technical definition of the client/server protocol is stated in terms of a relationship between software processes, a more popular definition describes the technical architecture that supports the software. Client/server architecture provides

an opportunity to distribute an application across two or more computers that can be used most effectively to deliver departmental and enterprise-wide systems solutions.

Distributed architecture is based on a network model in which processes can be distributed on any processor and any two individual nodes of the network are in a client/server relationship with any number of intervening middle layers. The heart of distributed architecture is based on the client/server pattern. The additional complexity comes from the design of the components for the middle layer, referred to as "middleware". Distributed application development takes a concurrent rather than sequential development approach, effectively using iteration to show progress and manage the risks. This provides a basis for more rapid development, with smooth transitioning from one stage of a project to the next as well as continuous delivery of staged results that are of value in the total solution. The use of this approach implies iteration with a checkpoint at the conclusion of each stage to validate the quality of stage results, as well as the scope of the next stage. In addition to concurrency across stages, a release-based strategy ensures that there are short intervals between incremental releases of tangible results. This ensures that the direction of the project can be adjusted dynamically to accommodate critical events and needs.

From the above discussion, it is clear that the development process of distributed software poses many difficult challenges before the programmer. Moreover, the complexity increases by many folds, when distributed object-oriented programs are concerned. Since most efforts are put in debugging, testing and maintenance of the software being developed, slicing technique can be used effectively to ease the burden.

Keeping the above identified objectives in mind, in this chapter we propose an algorithm for computing dynamic slices of distributed Java programs. In this chapter, we have extended the proposed *CGCA* for computing dynamic slices of distributed object-oriented programs. In the proposed algorithm, we have considered the features to handle the distributed nature of Java to give the impression of graph coloring technique. But in order to achieve the goal, we have contradicted some key constraints so as to make the process faster.

6.1 Basic Concepts and Notations

The concurrency model of Java has been discussed earlier. There are several methods i.e. *wait(), notify(), and notifyall ()* which help in *synchronization* among different threads. Java also provides methods to handle message passing among different threads. When a thread needs to send a message to another thread, it calls the method *getOutputStream()*. Similarly, to receive a message, the receiving thread calls the method *getInputStream()*. Java provides *sockets* to support distributed programming among the component programs running on different machines. A client program must specify the *ip* address and the *port number* of a sever program with which it wants to communicate.

A distributed object-oriented program P (p_1,p_2,...,p_n) is a collection of concurrent individual programs p_i such that each program p_i may communicate with other programs through the reception and transmission of messages. The individual programs p_i are known as component programs. We assume asynchronous send and synchronous receive message passing among component programs. However, other models can easily be considered through minor alterations to our proposed algorithm. Each component program may contain multiple threads. We assume use of *sockets* for message passing among threads of different component programs, and assume use of shared objects for message passing among different threads within a single component program.

6.2 Intermediate Representation

In order to compute the dynamic slice of a distributed object-oriented programs efficiently, the *Distributed Program Dependence Graph* (DPDG) [88] is taken as the intermediate representation.The reason behind choosing DPDG is that, it successfully addresses the problem that arises because of the communication dependence, which exists among threads running on different machines.

The CSDG can't be used here, since concurrency may exist among different threads running on different machines. A *getInputStream()* call executed on one machine, might

have a pairing *getOutputStream()* on some remote machine. To represent this aspect, a logical node called C-node is used. It may be noted that the number of C-nodes in the DPDGs of a distributed Java program, equals the number of *getInputStream()* calls present in the program.

C-Node: Let G_{D_1} and G_{D_2} be the DPDGs of two component programs $P1$ and $P2$ respectively. A C-Node represents a logical connection of the node y of DPDG G_{D_1} with the node x of the remote DPDG G_{D_2} . Node x represents the pairing of getOutputStream() with a getInputStream() call at node y. Node y is M-Communication dependent on node x.

The C-nodes maintain the logical connectivity among DPDGs representing different component programs. We therefore call them logical nodes. A C-node does not represent any specific statement in the source code of a component program.

In the DPDG, for a *getInputStream()* node x, the corresponding C-node is represented as *C(x)*. We can define the DPDG of a distributed program $P(p_1, p_2, \ldots, p_n)$ as a directed graph (N_{D_i}, E_{D_i}), where where each node n represents a statement in p_i. For $x, y \in N_{D_i}$, $(y, x) \in E_{D_i}$, anyone of the following must hold true:

1. If y is control dependent on x, then such an edge is called a control dependence edge. In Fig. 6.3 and Fig. 6.4, the edges (24, 25) and (30, 31) represent the control dependence edges.

2. If y is data dependent on x, then such an edge is called a data dependence edge. In Fig. 6.3 and Fig. 6.4, the edges (26, 27) and (59, 60) represent the control dependence edges.

3. If y is thread dependent on x, then such an edge is called a thread dependence edge. In Fig. 6.3 and Fig. 6.4, the edges (6, 7) and (24, 27) represent the control dependence edges.

4. If y is synchronization dependent on x, then such an edge is called a synchronization dependence edge. In Fig. 6.4, the edges (5, 11) and (7, 9) represent the control dependence edges.

5. If y is communication dependent on x, then such an edge is called a communication dependence edge. In Fig. 6.4, the edges (25, c(25)) represent the control dependence edge.

A Distributed Program Dependence Graph (DPDG) captures the basic thread structure of a distributed Java program component including it's run-time behavior. Hence, a DPDG represents dynamic thread creation, synchronization of threads, and inter-thread communication using shared objects and message passing. This graph contains the information available from other remote slicers by having additional logical nodes (C-nodes). More about DPDG can be found in [88].

```
1 class clthd extends Thread {
2       BufferedReader rcvmsg;
3       PrintWriter sendmsg;
4       BufferedReader in = new BufferedReader (
                        new InputStreamReader (system.in));
5       Socket socket;
6       Public void run(){
7               socket = new Socket ("10.5.18.49", 1500);
8               sendmsg = new PrintWriter (socket. getOutputStream ());
9               rcvmsg = new BufferedReader (
                new InputStreamReader (socket.getInputStream()));
10              String s = in.readLine();
11              int x = Integer.parseInt (s);
12              int z,q,y=15;
13              if (x>y)
14                      z = x – y ;
                else
15                      z = x + y ;
16              sendmsg.println (z);
17              System.out.println ("Value of z is"+z);
18              String msgserver = rcvmsg.readLine ();
19              int p = Integer.parseInt (msgserver);
20              if (p>x)
21                      q = p – x ;
                else
22                      q = p + x ;
23              System.out.println ("Total is"+q);
        } }
24 Public class client {
25              Public static void main (String s [ ]) {
26                              clthd t = new clthd ( );
27                              t.start ();
                }
    }
```

Figure 6.1: An example client program

```
1. class shar{
2.    int s;
3.    boolean flag=true;
4.    synchronized public void put(int c) {
5.       s=c;
6.       notify();
7.       flag=false;
      }
8.    synchronized public int get() {
9.              if(flag==true)
10.                 wait();
11.             return s;
       }
     }
12. class thread1 extends Thread {
13.             BufferedReader rcvmsg;
14.             PrintWriter sendmsg;
15.             Socket serv_socket;
16.             int b,c;
17.             shar obj; // declaration for shared object
18.             BufferedReader o=new BufferedReader(
                        new InputStreamReader(System.in));
19.             public thread1(Socket req,BufferedReader in,
                          PrintWriter out,shar ob) {
20.                       serv_socket=req;
21.                       sendmsg=out;
22.                       rcvmsg=in;
23.                       obj=ob;
                }
24.             public void run() {
25.                       String msg=rcvmsg.readLine(); // receiving message from client prog
26.                       int a=Integer.parseInt(msg);
27.                       System.out.println("received from client is: "+a);
28.                       String mss=o.readLine();
29.                       int b=Integer.parseInt(mss);
30.                       if(a>b)
31.                            c=a-b;
                          else
32.                            c=a+b;
33.                       obj.put(c); // sending the value of c to thread2
34.                       System.out.println("thd1: "+c);
              } }
35. class thread2 extends Thread {
36.             BufferedReader rcvmsg;
37.             PrintWriter sendmsg;
38.             Socket serv_socket;
39.             shar obj; // declaration for shared object
40.             BufferedReader o=new BufferedReader(
                          new InputStreamReader(System.in));
41.             public thread2(Socket req,BufferedReader in,
                          PrintWriter out,shar ob) {
42.                       serv_socket=req;
43.                       sendmsg=out;
44.                       rcvmsg=in;
45.                       obj=ob; }
46.             public void run() {
47.                       int e,g,f=10;
48.                       e=obj.get(); // receiving the value from thread1
49.                       if(e>f)
50.                            g=e-f;
                          else
51.                            g=e+f;
52.                       sendmsg.println(g); } } // sending value of g to client
53. public class syn_server {
54.             public static void main(String args[]) {
55.                       ServerSocket serv_socket;
56                        BufferedReader rcvmsg;
57.                       PrintWriter sendmsg;
58.                       shar obj=new shar();
59.                       serv_socket=new ServerSocket(1500); // creating a new port
60.                       Socket socket=serv_socket.accept(); // accepts client
61.                       sendmsg=new PrintWriter(socket.
                                    getOutputStream(),true); // declaration for sendmsg
62.                       rcvmsg=new BufferedReader(new
                                    InputStreamReader(socket.getInputStream())); // declaration for rcvmsg
63.                       thread1 t1=new thread1(socket,input,output,obj);
64.                       thread1 t2=new thread2(socket,input,output,obj);
65.                       t1.start();
66.                       t2.start();
              }
     }
```

Figure 6.2: An example server program

6.3 Dynamic Slicing of Distributed Object-Oriented Programs using CGC Algorithm

In this section, we first briefly explain our CGC Algorithm to compute the dynamic slice of distributed object-oriented programs. Subsequently, we illustrate the working of our algorithm through an example.

6.3.1 An Overview of CGC Algorithm

graph coloring is a technique, which is used to assign different colors to two adjacent nodes sharing the same edge. It is also popularly known as vertex coloring problem. Vertex coloring is a proven technique, which has seen wide application in the field of computer science. Hence as per the definition, the coloring of a graph $G = (V, E)$ is a mapping $c : V \rightarrow S$, where s is a finite set of colors, such that $(V, W) \in E$, then $c(V) = c(W)$. The number of colors to be used is known as chromatic number represented as $\chi(G)$.

As discussed earlier, program slice can be computed efficiently from a program dependence graph (PDG). So, vertex coloring can be used effectively to compute slice from a PDG. But graph coloring as it is, will not be helpful to compute slice, because it is a NP- problem. Hence to speed up the process, we have contradicted two constraints in the traditional vertex coloring technique:

- It is assumed that "two vertices sharing the same edge, can have the same color".
- The chromatic number of the graph is taken as one i.e. $\chi(G) = 1$.

6.3.2 Extending State Restriction To Handle Dependence

Binkley et al. [20] have given a universal view of state restriction. Since, we are going to compute the dynamic slice of a distributed object-oriented program, we have to take into consideration the issues that may arise due to control dependence (cd), data dependence (dd), synchronization dependence (sd) or communication dependence (cmd). Hence as per our requirements we have modified the definition of state restriction given in Eq. 2.1

Hence in order to capture the proper state restriction we modify the original definition as below:

"If σ the given state and V be the set of variables then σ is state restricted by V when there exists either control dependence, data dependence, synchronization dependence or communication dependence."

Hence, in program projection theory notation, it can be written as:

$$(\sigma \upharpoonright V)_{cd||dd||sd||cmd} = \begin{cases} \sigma x & \text{if } x \in V \\ \perp & \text{otherwise} \end{cases} \tag{6.1}$$

6.4 Slicing of Distributed Object-Oriented Programs Using DPDG

In Fig. 6.1 and Fig. 6.2, we have shown an example distributed Java program taken from [88] for which we want to compute the slice using our proposed new algorithm and the DPDG. The algorithm is modeled using the program projection theory.

We first provide a brief overview of our dynamic slicing algorithm. First, we construct the DPDG of each component program statically once. During the execution of a component program, we color a node when it is found to state restrict the slicing criterion depending upon the existence of the dependence. We have considered different types of dependences while computing slice i.e. control dependence, data dependence, thread dependence, synchronization dependence and communication dependence. Our proposed algorithm supports inter-thread synchronization and communication across different machines.

Contradictory graph coloring algorithm for dynamic slicing of distributed object-oriented programs

Input: Set of nodes of DPDG ($n_1, n_2, n_3, \ldots, n_m$), n_c being the node representing the slicing criterion, current state σ of slicing criterion, set of variables V ($x_1, x_2, \ldots, x_j$)

output: Colored nodes showing the slice on the CSDG.

1. Draw the DPDG statically once.

2. Set the chromatic number χ(DPDG)=1 as we are going to use one color to compute the slice.

3. Traverse backward the graph DPDG.

while(traverseNode ≤ m)
{
n_i = traverseNode ;
if (($\sigma \uparrow x_j)_{cd}$ for all $x_j \in V$) , **then**
color the node reached n_r from n_c during traversal
else if (($\sigma \uparrow x_j)_{dd}$ for all $x_j \in V$) , **then**
{
if the node reached n_r is already colored, **then**
go to label1
else
color the node reached n_r.
}
Label1:**else if**(($\sigma \uparrow x_j)_{sd||cmd}$ for all $x_j \in V$), **then**
{
if the node reached n_r is already colored, **then**
{
traversedNode=+1;
continue;
}
else
{

color the node n_r reached;
traversedNode=+1;
}
}
}

4. Look up the nodes which are colored during the graph traversal representing the slice.

6.5 Working of The Algorithm

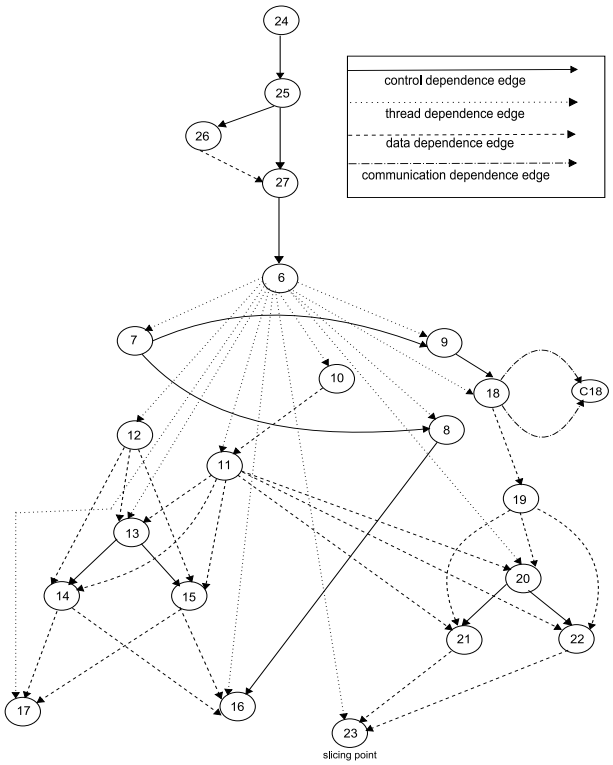

Figure 6.3: The DPDG for the client program given in Fig. 6.1

We explain the working of the algorithm with the help of the program given in Fig. 6.1 and Fig. 6.2. For the given client-server program, the DPDG is drawn as described in. Then the DPDG is traversed backward and the CGC Algorithm is applied to update the graph. The DPDG is updated by assigning color, which state restrict the slicing criterion by either control dependence(cd), data dependence(dd), synchronization dependence(sd) or communication dependence(cmd). Below the working of

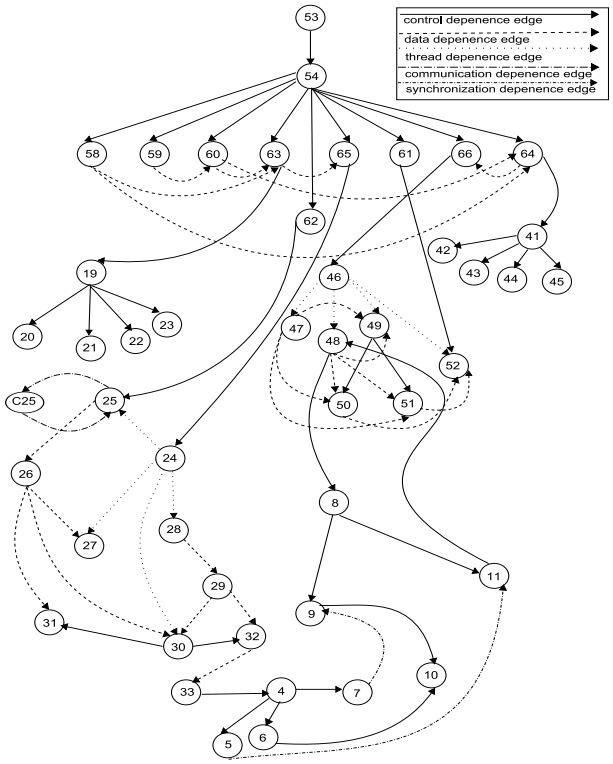

Figure 6.4: The DPDG for the server program given in Fig. 6.2

the algorithm w.r.t. the slicing criterion $< 23, q >$ for the input value of $s = 20$ of client program and $b = 2$ of the server program is illustrated. During the traversal, the nodes of the DPDG are traversed in a backward manner. The algorithm has colored the nodes 6 and 10 as synchronization dependence exists between the statements. Similarly, communication dependence exists between the nodes 9 and 7; also between 11 and 5, for which these nodes are colored.

6.6 Implementation Results

We have implemented our algorithm using Java and ANTLR [59]. To construct the intermediate representation, the DPDG, ANTLR (Another Tool for Language Recognition) is used to parse the example program taken into consideration. ANTLR allows one to implement the ANTLR defined grammar by automatically generating lexers and parsers in Java and other supported languages. An adjacency matrix is used to store the nodes of the DPDG of the program. A special integer value is used to represent the colored node, which may state restrict the slicing criterion. The execution time for the

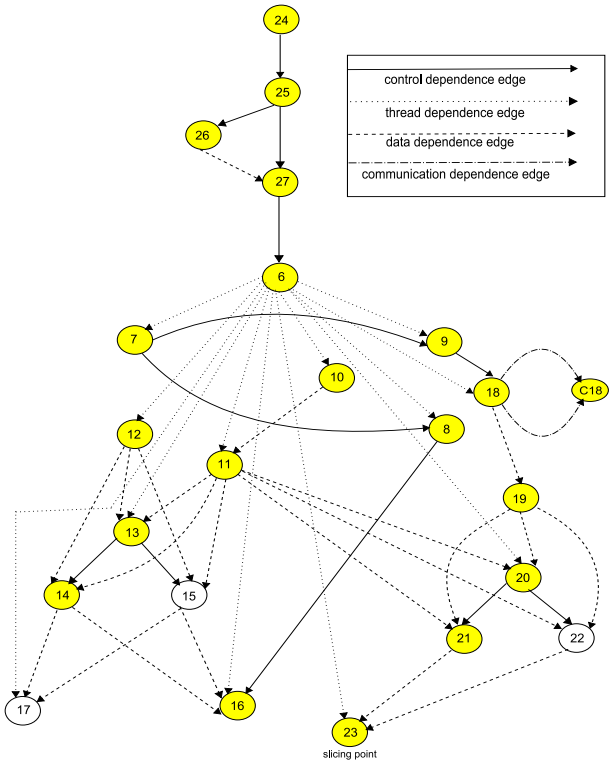

Figure 6.5: Updated DPDG of client program

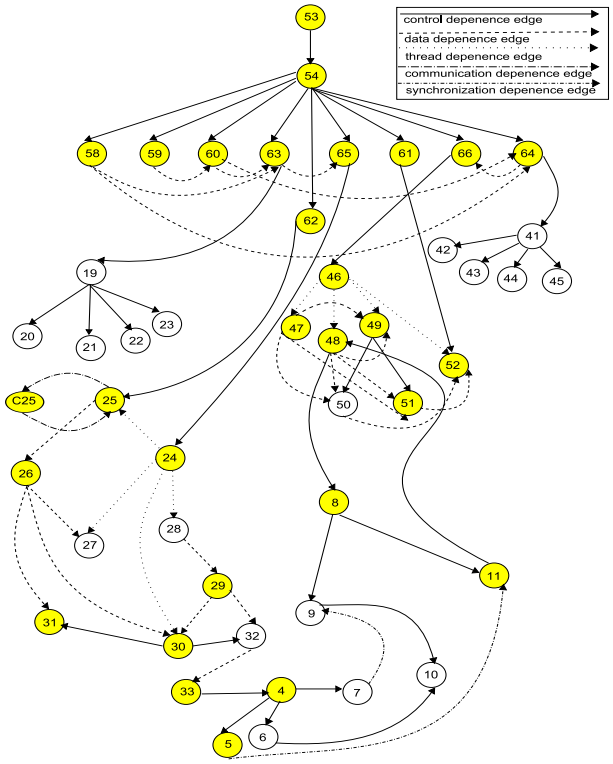

Figure 6.6: Updated DPDG of server program

CGCA is studied and the performance result is compared with that of DDS algorithm and is given in Fig. 6.7.

Table 6.1: Comparison of average run-time between DDS and CGCA for distributed object-oriented programs

Sl. No.	Program Size (No. of stmt.)	Average Runtime (in Sec.)	
		DDS	CGCA
1.	93	0.12	0.09
2.	243	0.34	0.28
3.	328	0.42	0.35
4.	468	0.53	0.45
5.	550	0.59	0.50
6.	640	0.68	0.58
7.	725	0.78	0.67
8.	846	0.91	0.78
9.	918	1.06	0.92

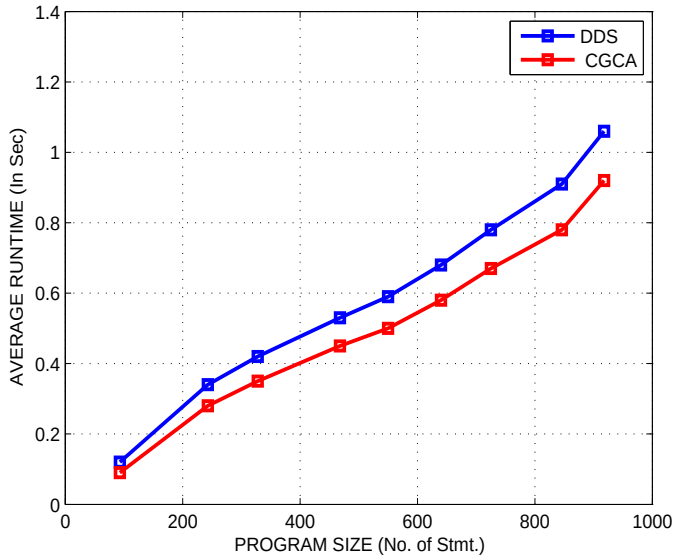

Figure 6.7: Comparison of average runtime between CGCA and DDS algorithm

In distributed dynamic slicing (DDS), the distributed program dependence graph (DPDG) is updated by marking and unmarking of the nodes. When the dependence exists the, the edges are marked and when the dependence ceases to exist, the edges are unmarked. Hence the process of marking and umarking consumes considerable amount of time in updating the graph.

On the other hand, CGCA works by coloring the nodes of the DPDG. Apart from dependence analysis, we have taken the state of the slicing criterion. Our algorithm

colors a node, if and only if it is found to state restrict the slicing criterion. Hence unlike the DDS algorithm, in our CGC algorithm time is consumed only to color the nodes representing the slice.

6.7 Comparison With Related Work

Korel and Ferguson [63] have proposed a dynamic slicing algorithm for Ada programs. According to their approach, each process generates a complete execution trace. Having done the dependence analysis, slices are computed from the trace file. The disadvantage of this approach is that, the entire process was inefficient because of the use of trace file. We have computed the dynamic slice of distributed object-oriented programs. Moreover, our algorithm is efficient because, trace file is not used during the process of slice computation.

Duesterwald et al. [37] have presented a parallel algorithm for computing dynamic slice of procedural distributed programs. They have proposed a new graph called *dynamic dependence graph* (DDG) as the intermediate representation. By solving it as a graph reachability problem, slices are computed from the DDG. They used both static and dynamic information to compute slice. The disadvantage of this algorithm is that, it can't be applied to programs which send and receive messages asynchronously. Our algorithm deals with computing the dynamic slice of distributed object-oriented programs. The proposed contradictory graph coloring algorithm works iteratively by coloring the nodes, it is more time efficient than the previous one. Our algorithm can be applied to compute the dynamic slice of object-oriented programs, which send and receive messages asynchronously.

Li et al. [76] have given the notion of predicate based dynamic slicing algorithm. Their algorithm is based on partially ordered multi-set (POMSET) model. Unlike traditional slicing criteria, a predicate focuses on those parts of the program, which may influence it. Their proposed algorithm was able to handle message passing systems. But they haven't considered object-oriented features. Our algorithm is specifically designed for computing dynamic slices of object-oriented programs. Moreover, the

slices computed using CGCA are more useful in Client-Server applications.

Mohapatra et al. [88] have proposed a marking based dynamic slicing technique for computing slice of a distributed object-oriented program. They have proposed a new graph called distributed program dependence (DPDG) as the intermediate program dependence representation. Their proposed algorithm also known as distributed dynamic slicing (DDS) algorithm works by marking the edges when dependencies arise and unmarking the edges when dependencies cease to exist. The advantage of the algorithm is that, it is able to compute precise dynamic slice. But it takes considerable amount of time to compute the slice as the process of marking and unmarking consumes significant amount of time and makes the entire process slower. But our algorithm works by coloring the nodes of the DPDG, which is more efficient than the DDS algorithm, because time is not consumed in unmarking the nodes of DPDG as happened in the DDS algorithm [88].

6.8 Conclusion

In this chapter, we have proposed a novel technique for computing dynamic slices of distributed Java programs. We have used the DPDG as the intermediate representation for our slicing algorithm. We have used graph coloring technique on the DPDG to compute the dynamic slice. Our algorithm addresses the concurrency issues of Java programs while computing the dynamic slices. It also handles the communication dependency arising due to objects shared among threads on same machine and message passing among threads on different machines. While computing slice the state restriction of the slicing criterion is taken into consideration in addition to the dependence analysis. So our proposed algorithm is more efficient than the existing algorithms.

Chapter 7

Conclusion

The work in this thesis, primarily focuses on computing the dynamic slice of object-oriented programs. A novel algorithm for dynamic slicing of object-oriented programs has been proposed, which incorporates graph coloring technique. The work reported in this thesis is summarized in this chapter. Section 6.1 deals with our contribution and Section 6.2 provides some scope for further development.

7.1 Contributions

There are three major contributions, which are: dynamic slicing of object-oriented programs, dynamic slicing of concurrent object-oriented programs and dynamic slicing of distributed object-oriented programs.

In order to compute dynamic slice of object-oriented programs, the proven system dependence graph (SDG) is taken as the intermediate representation. The SDG successfully represents different object-oriented features such as class representation, method invocation, inheritance etc. Then, a graph coloring algorithm is proposed to compute the dynamic slice. But in order to compute the dynamic slice efficiently, the constraints in the traditional graph coloring technique have been removed.The chromatic number is taken as 1. In addition to it, the state restriction of the slicing criterion is taken into consideration. Simulation results have revealed that our proposed algorithm is more time efficient than the existing one. We have named our algorithm *Contradictory Graph Coloring Algorithm* abbreviated as *CGCA*.

The second contribution made towards the dynamic slicing of concurrent object-oriented programs. Due to the presence of inter-thread synchronization and communi-

cation, some control and data flows in the threads are interdependent. To capture this aspect, the proposed *CGCA* is extended to compute the dynamic slice of concurrent object-oriented programs. The CSDG is taken as the intermediate representation. Then the proposed CGCA is applied to compute the dynamic slice from the CSDG. The run time of CGCA was compared with MBDS algorithm. It was found that the proposed CGCA is more efficient than the MBDS algorithm.

Further, the proposed CGCA is extended to compute the dynamic slice of distributed object-oriented programs, which constitutes our third contribution. In distributed object-oriented programs, the components program communicate with each other by using sockets. To capture all the aspects of distributed object-oriented programs, we have chosen the DPDG as the intermediate program representation. The run-time of the CGCA for dynamic slicing of distributed object-oriented programs was compared with MBDS algorithm, which showed that the proposed CGCA to be more efficient than the MBDS algorithm.

7.2 Future Work

- In our proposed work, composite data like arrays, lists are not considered. The CGCA can be easily extended to handle these above said composite data.
- Generic programming has been available in J2SE 5.0 version onwards. Our proposed algorithm can be easily modified to compute dynamic slice of generic object-oriented programs.
- The proposed algorithm can be effectively used for test case generation of object-oriented programs.
- Moreover, the proposed contradictory graph coloring algorithm presented in this thesis is a very basic one. It can be made more efficient by applying any off-the-shelf existing optimization technique like heuristic method meta-heuristic method, probabilistic method etc.

Bibliography

[1] Agrawal, H., DeMillo, R. A., and Spafford, E. H., "Dynamic slicing in the presence of unconstrained pointers," in *Proceedings of the ACM Fourth Symposium on Testing, Analysis and Verification (TAV4)*, pp. 60 – 73, 1991.

[2] Agrawal, H., DeMillo, R. A., and Spafford, E. H., "Debugging with dynamic slicing and backtracking," *Software Practice and Experience*, vol. 23, no. 6, pp. 589 – 616, 1993.

[3] Agrawal, H. and Horgan, J., "Dynamic program slicing," in *Proceedings of the ACM SIGPLAN'90 Conference on Programmimg Lanuages Design and Implementation*, (White Plains, New York), pp. 246 – 256, SIGPLAN Notices, Analysis and Verification, 1990.

[4] Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[5] Al Dallal, J., "Using computing-all-slices algorithm in measuring functional cohesion," in *SE'07: Proceedings of the 25th conference on IASTED International Multi-Conference*, (Anaheim, CA, USA), pp. 198–203, ACTA Press, 2007.

[6] Andres, S. D., "Directed defective asymmetric graph coloring games," *Discrete Appl. Math.*, vol. 158, no. 4, pp. 251–260, 2010.

[7] Andrews, G. R., *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.

[8] Andrews, G. R. and Schneider, F. B., "Concepts and notations for concurrent programming," *ACM Computing Surveys*, vol. 15, pp. 3 – 43, 1983.

[9] Awad, M. and Ziegler, J., "A practical approach to the design of concurrency in object-oriented systems," *Software Practice and Experience*, vol. 27, pp. 1013 – 1034, 1997.

[10] Ball, T., *The Use of Control Flow and Control Dependence in Software Tools*. PhD thesis, Department of Computer Science, University of Wisconsin-Madison, 1993.

[11] Barraclough, R. W., Binkley, D., Danicic, S., Harman, M., Hierons, R. M., Kiss, A., Laurence, M., and Ouarbya, L., "A trajectory-based strict semantics for program slicing," *Theoretical Computer Science*, vol. 411, no. 11-13, pp. 1372–1386, 2010.

[12] Beck, J. and Eichman, D., "Program and interface slicing for reverse engineering," in *Proceedings of the IEEE/ACM Fifteenth International Conference on Software Engineering*, pp. 509 – 518, 1993.

[13] Ben Ari, M., *Principles of Concurrent and Distributed Programming*. Prentice Hall, 1990.

[14] Bieman, J. M. and Ott, L. M., "Measuring functional cohesion," *IEEE Transactions on Software Engineering*, vol. 20, pp. 644 – 657, August 1994.

[15] Binkley, D., "Using semantic differencing to reduce the cost of regression testing," in *Proceedings of the IEEE Conference on Software Maintenance*, pp. 41 – 50, November 1992.

[16] Binkley, D., "The application of program slicing to regression testing," *Information and Software Technology*, vol. 40, no. 11-12, pp. 583 – 594, 1998. Special Issue on Program Slicing.

[17] Binkley, D., "Computing amorphous program slices using dependence graphs and a data flow model," in *Proceedings of the ACM Symposium on Applied Computing*, ACM Press, 1999.

[18] Binkley, D. and Gallagher, K. B., "Program slicing," *Advances in Computers*, vol. 43, 1996. Academic Press, San Diego, CA.

[19] Binkley, D., Horwitz, S., and Reps, T., "Program integration for languages with procedure calls," *ACM Transactions on Software Engineering and Methodology*, vol. 4, no. 1, pp. 3 – 31, 1995.

[20] Binkley, D., Danicic, S., Gyimothy, T., Harman, M., Kiss, A., and Korel, B., "A formalisation of the relationship between forms of program slicing," *Science of Computer Programming*, vol. 62, no. 5, pp. 228 – 252, 2006.

[21] Binkley, D., Harman, M., Hassoun, Y., Islam, S., and Li, Z., "Assessing the impact of global variables on program dependence and dependence clusters," *Journal of System and Software*, vol. 83, no. 1, pp. 96–107, 2010.

[22] Canfora, G., Cimitile, A., and Lucia, A. D., "Conditioned program slicing," *Information and Software Technology*, vol. 40, pp. 595 – 607, 1998.

[23] Carlin, P., Chandy, M., and Kesselman, C., "The compositional c++ language definition," tech. rep., California Institute of Technology, Department of Computer Science, February 1993.

[24] Chen, J. T., Wang, F. J., and Chen, Y. L., "Slicing object-oriented programs," in *Proceedings of the APSEC'97*, (Hongkong, China), pp. 395 – 404, December 1997.

[25] Chen, Z. and Xu, B., "Slicing concurrent java programs," *ACM SIGPLAN Notices*, vol. 36, pp. 41 – 47, 2001.

[26] Chen, Z. and Xu, B., "Slicing concurrent java programs," *ACM SIGPLAN Notices*, vol. 36, pp. 33 – 40, 2001.

[27] Chen, Z., Xu, B., and Yang, H., "Test coverage analysis based on program slicing," in *Proceedings of IRI*, pp. 559 – 565, 2003.

[28] Chen, Z., Xu, B., and Zhao, J., "An overview of methods for dependence analysis of concurrent programs," *ACM SIGPLAN Notices*, vol. 37, no. 8, pp. 45 – 52, 2002.

[29] Chen, Z., Zhou, Y., Xu, B., Zhao, J., and Yang, H., "A novel approach for measuring class cohesion based on dependence analysis," in *Proceedings of International Conference on Software Maintenance*, pp. 377 – 384, IEEE Press, 2002.

[30] Cheng, J., "Slicing concurrent programs - a graph theoretical approach," in *Automated and Algorithmic Debugging, AADEBUG'93* (LNCS, ed.), pp. 223 – 240, Springer-Verlag, 1993.

[31] Cheng, J., "Dependence analysis of parallel and distributed programs and its applications," in *International Conference on Advances in Parallel and Distributed Computing*, pp. 370 – 377, 1997.

[32] Cheng, J., "Uncertainty problem in dynamic slicing of concurrent programs," in *ICESS '09: Proceedings of the 2009 International Conference on Embedded Software and Systems*, (Washington, DC, USA), pp. 241–248, IEEE Computer Society, 2009.

[33] Choi, J. D., Miller, B., and Netzer, R., "Techniques for debugging parallel programs with flowback analysis," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 491 – 530, 1991.

[34] Danicic, S., Harman, M., and Sivagurunathan, Y. A., "parallel algorithm for static program slicing," *Information Processing Letters*, vol. 56, pp. 307 – 313, 1995.

[35] Deo, N., *Graph Theory With Applications to Engineering and Computer Science*. Prentice-Hall, 1974.

[36] Dhamdhere, D. M., Gururaja, K., and Ganu, P. G., "A compact execution history for dynamic slicing," *Information Processing Letters*, vol. 85, pp. 145 – 152, 2003.

[37] Duesterwald, E., Gupta, R., and Soffa, M. L., "Distributed slicing and partial reexecution for distributed programs," in *Fifth Workshop on Languages and Compilers for Parallel Computing* (LNCS, ed.), (New Haven Connecticut), pp. 329 –337, Springer-Verlag, August 1992.

[38] Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and it's use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319 – 349, 1987.

[39] Field, J., Ramalingam, G., and Tip, F., "Parametric program slicing," in *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages*, (San Francisco, CA, USA), pp. 379 – 392, ACM, 1995.

[40] Forgacs, I. and Bertolino, A., "Feasible test path selection by principal slicing," in *Proceedings of 6th Euoropean Software Engineering Conference*, September 1997.

[41] Gallagher, K. and Lyle, J., "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 751 – 761, 1991.

[42] Garg, V. K. and Mittal, N., "On slicing a distributed computation," in *Proceedings of 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 322 – 329, 2001.

[43] Goel, A., RoyChoudhury, A., and Mitra, T., "Compactly representing parallel program executions," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 191 – 202, 2003.

[44] Goswami, D. and Mall, R., "Fast slicing of concurrent programs," in *Sixth International Conference on High Performance Computing (HiPC)* (LNCS, ed.), pp. 38 – 42, Springer-Verlag, December 1999.

[45] Goswami, D. and Mall, R., "Dynamic slicing of concurrent programs," in *Seventh International Conference on High Performance Computing (HiPC)* (LNCS, ed.), pp. 17 – 26, Springer-Verlag, December 2000.

[46] Goswami, D., Mall, R., and Chatterjee, P., "Static slicing in unix process environment," *Software Pracice and Experience*, vol. 30, pp. 17 – 36, 2000.

[47] Gupta, R., Harrold, M. J., and Soffa, M. L., "Program slicing-based regression testing techniques," *Journal of Software Testing, Verification and Reliability*, vol. 6, 1996.

[48] Gupta, R. and Soffa, M. L., "Hybrid slicing: An approach for refining static slices using dynamic information," in *Proceedings of ACM SIGSOFT*, pp. 29 – 40, 1995.

[49] Hall, R., "Automatic extraction of executable program subsets," *Automated Software Engineering*, vol. 2, pp. 33 – 53, 1995.

[50] Hammer, C. and Snelting, G., "An improved slicer for java," in *Proceedings of PASTE*, pp. 107 – 112, 2004.

[51] Harman, M., "Conditioned slicing supports partition testing," *Journal of Software Testing, Verification and Reliability*, vol. 12, pp. 23 – 28, 2002.

[52] Harman, M., Binkley, D., and Danicic, S., "Amorphous program slicing," *Journal of Systems and Software*, vol. 68, pp. 45 – 64, 2003.

[53] Harman, M. and Danicic, S., "Using program slicing to simplify testing," *Journal of Software Testing, Verification and Reliability*, vol. 5, 1995.

[54] Harman, M. and Hierons, R. M., "An overview of program slicing," *Software Focus*, vol. 2, pp. 85 – 92, 2001.

[55] Harman, M., L. Hu, M. M., Zhang, X., Binkley, D., and Danicic, S., "Syntax-directed amorphous slicing," *Automated Software Engineering*, vol. 11, pp. 27 – 61, 2004.

[56] Harrold, M. J. and Rothermel, G., "Performing data flow testing on classes," in *Second ACM SIGSOFT Symposium on the Foundation of Software Engineering*, pp. 154 – 163, December 1994.

[57] Horwitz, S., Prins, J., and Reps, T., "Integrating noninterfering versions of programs," *ACM Transactions on Programming Languages and Systems*, vol. 11, pp. 345 – 387, July 1989.

[58] Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26 – 61, 1990.

[59] http://www.antlr.org/, A.

[60] Jaroenpiboonkit, J. and Suwannasart, T., "Finding a test order using object-oriented slicing technique," in *APSEC '07: Proceedings of the 14th Asia-Pacific Software Engineering Conference*, (Washington, DC, USA), pp. 49–56, IEEE Computer Society, 2007.

[61] Kamkar, M., *Inter Procedural Dynamic Slicing with Applications to Debugging and Testing*. PhD thesis, Linkoping University, Sweden, 1993.

[62] Kamkar, M. and Krajina, P., "Dynamic slicing of distributed programs," in *International Conference on Software Maintenance*, pp. 222 – 229, IEEE CS Press, October 1995.

[63] Korel, B. and Ferguson, R., "Dynamic slicing of distributed programs," *Applied Mathematics and Computer Science*, vol. 2, pp. 199 – 215, 1992.

[64] Korel, B. and Laski, J., "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155 – 163, 1988.

[65] Korel, B. and Rilling, J., "Dynamic program slicing methods," *Information and Software Technology*, vol. 40, pp. 647 – 659, 1998.

[66] Krinke, J., "Static slicing of threaded programs," *ACM SIGPLAN Notices*, vol. 33, pp. 35 – 42, April 1998.

[67] Krinke, J., "Context-sensitive slicing of concurrent programs," in *Proceedings of ACM SIGSOFT Software Engineering Notes*, pp. 178 – 187, 2003.

[68] Krishnaswamy, A., "Program slicing: An application of program dependency graphs," tech. rep., Clemson University, Department of Computer Science, August 1994.

[69] Kuhn, F., "Weak graph colorings: distributed algorithms and applications," in *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, (New York, NY, USA), pp. 138–144, ACM, 2009.

[70] Kung, D., Gao, J., Hisa, P., and Toyoshima, Y., "Change impact identification in object-oriented software maintenance," in *Proceedings of International Conference on Software Maintenance*, pp. 202 – 211, September 1994.

[71] Kung, D., Gao, J., Hisa, P., and Toyoshima, Y., "Firewall regression testing and software maintenance of object-oriented systems," *Journal of Object-Oriented Programming*, 1994.

[72] Kung, D., Gao, J., Hisa, P., Toyoshima, Y., and Chen, C., "Design recovery for software testing of object-oriented programs," in *Working Conference on Reverse Engineering*, pp. 202 – 211, May 1993.

[73] Lanubile, F. and Visaggio, G., "Extracting reusable functions by flow graph-based program slicing," *IEEE Transactions on Software Engineering*, vol. 23, pp. 246 – 259, 1997.

[74] Larson, L. D. and Harrold, M. J., "Slicing object-oriented software," in *Proceedings of the 18th International Conference on Software Engineering*, (German), March 1996.

[75] Levine, J. R., Mason, T., and Brown, D., *Lex and Yacc*. O'REILLY, 3rd edition ed., 2002.

[76] Li, H. F., Rilling, J., and Goswami, D., "Granularity-driven dynamic predicate slicing algorithms for message passing systems," *Automated Software Engineering*, vol. 11, pp. 63 – 89, 2004.

[77] Liang, D. and Larson, L., "Slicing objects using system dependence graphs," in *Proceedings of International Conference on Software Maintenance*, pp. 358 – 367, November 1998.

[78] Lucia, A. D., "Program slicing: Methods and applications," in *Proceedings of IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 142 – 149, 2001.

[79] Lucia, A. D., Harman, M., Hierons, R., and Krinke, J., "Union slices are not slices," in *Proceedings of the 7th European Conference on Software Maintenance and Re-engineering (CSMR)*, pp. 363 – 367, 2003.

[80] Lyle, J. R. and Weiser, M. D., "Automatic program bug location by program slicing," in *Proceedings of the second International Conference on Computers and Applications*, (Peking, China), pp. 877 – 882, 1987.

[81] Mall, R., *Fundamentals of Software Engineering*. India: Prentice Hall, 3rd edition ed., 2003.

[82] Malloy, B. A., McGregor, J. D., and Krishnaswamy, A., "An extensible program representation for object oriented software," in *Proceedings of ISFST*, pp. 105 – 112, 2004.

[83] Mills, A. J. S., *Antlr*. The University of Birmingham, 2002.

[84] Mittal, N. and Garg, V. K., "Computation slicing: Techniques and theory," in *Proceedings of Symposium on Distributed Computing*, 2001.

[85] Mittal, N. and Garg, V. K., "Computation slicing: Techniques and theory," tech. rep., The University of Texas at Austin, The Parallel and Distributed Systems Laboratory, Department of Electrical and Computer Engineering, 2001. TR-PDS-2001-02.

[86] Mohapatra, D. P., *Dynamic Slicing of Object-Oriented Programs*. PhD thesis, I.I.T. Kharagpur, May 2005.

[87] Mohapatra, D. P., Kumar, R., and Mall, R., "Computing dynamic slices of concurrent object-oriented programs," *Information and Software Technology*, vol. 47, no. 5, pp. 805 – 817, 2005.

[88] Mohapatra, D. P., Mall, R., Kumar, R., Kumar, D., and Bhasin, M., "Distributed dynamic slicing of java programs," *The Journal of Systems and Software*, vol. 79, pp. 1661 – 1678, 2006.

[89] Mund, G. B., Mall, R., and Sarkar, S., "An efficient dynamic program slicing technique," *Information and Software Technology*, vol. 44, pp. 123 – 132, 2002.

[90] Nanda, M. G. and Ramesh, S., "Slicing concurrent programs," in *ACM International Symposium on Software Testing and Analysis*, August 2000.

[91] Naor, M. and Stockmeyer, L., "What can be computed locally?," *SIAM journal of Computation*, vol. 24, no. 6, pp. 1259–1277, 1995.

[92] Naughton, P. and Schildt, H., *Java - The Complete Reference*. Mc GrawHill, 5th edition ed., 2006.

[93] Ohata, F., Hirose, K., Fujii, M., and Inoue, K., "A slicing method for object-oriented programs using lightweight dynamic information," in *APSEC '01: Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*, (Washington, DC, USA), p. 273, IEEE Computer Society, 2001.

[94] Ottenstein, K. and Ottenstein, L., "The program dependence graph in software development environment," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Develop Environments SIGPLAN Notices*, pp. 177 – 184, 1984.

[95] Qi, X. and Xu, B., "Dependence analysis of concurrent programs based on rechability graph and it's applications," in *Proceedings of International Conference on Computational Science*, pp. 405 – 408, 2004.

[96] Ranganath, V. P. and Hatcliff, J., "Slicing concurrent java programs using indus and kaveri," *International Journal of Software Tools Technology Transfer*, vol. 9, no. 5, pp. 489–504, 2007.

[97] Reps, T. and Rosay, G., "Precise interprocedural chopping," in *Proceedings of Third ACM Symposium on the Foundations of Software Engineering*, pp. 41 – 52, October 1995.

[98] Rilling, J., Li, H. F., and Goswami, D., "Predicate based dynamic slicing of message passing programs," in *Proceedings of IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 133 – 144, 2002.

[99] Sebastian Danicic, Mark Harman, John Howroyd, and Lahcen Ouarbya, "A non-standard semantics for program slicing and dependence analysis," *The Journal of Logic and Algebraic Programming*, vol. 72, pp. 191 – 206, July-August 2007.

[100] Singhal, M. and Sivaratri, N. G., *Advanced Concepts in Operating Systems - Distributed, Database, and Multiprocessor Operating Systems*. TATA McGRAW HILL, 2002.

[101] Song, Y. and Huynh, D., "Forward dynamic object-oriented program slicing," in *Application Specific Systems and Software Engineering and Technology (ASSET'99)*, IEEE CS Press, 1999.

[102] Tip, F., "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. 3, pp. 121 – 189, 1995.

[103] Tip, F., Choi, J. D., Field, J., and Ramalingam, G., "Slicing class hierarchies in c++," in *Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 179 – 197, 1996.

[104] Tonella, P., Antoniol, G., Fiutem, R., and Merlo, E., "Flow insensitive c++ pointers and polymorphism analysis and its application to slicing," in *Proceedings of 19th International Conference on Software Engineering*, pp. 433 – 443, May 1997.

[105] Umemori, F., Konda, K., Yokomori, R., and Inoue, K., "Design and implementation of bytecode-based java slicing system," in *Proceedings of IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 108 – 117, 2003.

[106] Wakinshaw, N., Roper, M., and Wood, M., "The java system dependence graph," in *Proceedings of IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 145 – 154, 2002.

[107] Wang, T. and RoyChoudhury, A., "Using compressed bytecode traces for slicing java programs," in *Proceedings of IEEE International Confrence on Software Engineering*, pp. 512 – 521, 2004.

[108] Wang, T. and Roychoudhury, A., "Dynamic slicing on java bytecode traces," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 2, pp. 1–49, 2008.

[109] Weeratunge, D., Zhang, X., Sumner, W. N., and Jagannathan, S., "Analyzing concurrency bugs using dual slicing," in *ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis*, (New York, NY, USA), pp. 253–264, ACM, 2010.

[110] Weiser, M., *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[111] Weiser, M., "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446 – 452, 1982.

[112] Weiser, M., "Reconstructing sequential behavior from parallel behavior projections," *Information Processing Letters*, vol. 17, no. 10, pp. 129 – 135, 1983.

[113] Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352 – 357, 1984.

[114] Xu, B. and Chen, Z., "Dynamic slicing object-oriented programs for debugging," in *International conference on source code analysis and manipulation (SCAM)*, pp. 115 – 122, 2002.

[115] Xu, B., Qian, J., Zhang, X., Wu, Z., and Chen, L., "A brief survey of program slicing," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 2, pp. 1 – 36, 2005.

[116] Xu, L., Xu, B., Chen, Z., Jiang, J., Chen, H., and Yang, H., "Regression testing for web applications based on slicing," in *Proceedings of 28th IEEE Annual International Computer Software and Applications Conference*, pp. 652 – 656, IEEE CS Press, 2003.

[117] Zeng, J., Soviani, C., and Edwards, S. A., "Generating fast code from concurrent program dependence graph," in *Proceedings of ACM LCTES*, pp. 175 – 181, 2004.

[118] Zhang, X., Gupta, R., and Zhang, Y., "Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams," in *International Conference on Software Engineering*, 2004.

[119] Zhang, X. and Gupta, R., "Cost effective dynamic program slicing," in *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, (New York, NY, USA), pp. 94–106, ACM, 2004.

[120] Zhang, Y., Xu, B., Shi, L., Li, B., and Yang, H., "Modular monadic program slicing," in *Proceedings of 28th IEEE Annual International Computer Software and Applications Conference*, pp. 66 – 71, IEEE CS Press, September 2004.

[121] Zhao, J., "Dynamic slicing of object-oriented programs," tech. rep., Tech. rep., Information Processing Society of Japan, May 1998.

[122] Zhao, J., "Multithreaded dependence graphs for cuncurrent java programs," in *Proceedings of the 1999 International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'99)*, 1999.

[123] Zhao, J., "Slicing concurrent java programs," in *Proceedings of the 7th IEEE International Workshop on Program Comprehension*, May 1999.

[124] Zhao, J., Cheng, J., and Ushijima, K., "Static slicing of concurrent object-oriented programs," in *20th IEEE Annual International Computer Software and Applications Conference*, pp. 312 – 320, August 1996.

[125] Zhao, J., Cheng, J., and Ushijima, K., "A dependence-based representation for concurrent object-oriented software maintenance," in *Proceedings of 2nd Euromicro Conference on Software Maintenance and Reengineering*, pp. 60 – 66, March 1998.

[126] Zhao, J. and Li, B., "Dependence based representation for concurrent java programs and it's application to slicing," in *Proceedings of ISFST*, pp. 105 – 112, 2004.